Efficient and High-quality Sparse Graph Coloring on the GPU*

Xuhao Chen† **Pingfan Li** **Canqun Yang**

April 2016
NUDT-CS-2016-003

College of Computer
National University of Defense Technology
Changsha, Hunan, China 410073

*The source code of this work can be found here: https://github.com/chenxuhao/csrcolor.

†Contact email: cxh.nudt@gmail.com

This research is supported by the National Science Foundation of China under grant No.61502514.

Keywords: Graph Coloring, GPU, Speculative Greedy

Abstract

Graph coloring has been broadly used to discover concurrency in parallel computing. To speedup graph coloring for large-scale datasets, parallel algorithms have been proposed to leverage modern GPUs. Existing GPU implementations either have limited performance or yield unsatisfactory coloring quality (too many colors assigned). We present a work-efficient parallel graph coloring implementation on GPUs with good coloring quality. Our approach employs the *speculative greedy* scheme which inherently yields better quality than the method of finding *maximal independent set*. In order to achieve high performance on GPUs, we refine the algorithm to leverage efficient operators and alleviate conflicts. We also incorporate common optimization techniques to further improve performance. Our method is evaluated with both synthetic and real-world sparse graphs on the NVIDIA GPU. Experimental results show that our proposed implementation achieves averaged $4.1\times$ (up to $8.9\times$) speedup over the serial implementation. It also outperforms the existing GPU implementation from the NVIDIA CUSPARSE library ($2.2\times$ average speedup), while yielding much better coloring quality than CUSPARSE.

1 Introduction

Graph processing algorithms are getting a growing research interest in the past decade. They are pervasively used in many application domains, such as scientific computing, social networks, simulations and bioinformatics. Parallelizing graph algorithms is challenging because of their inherent irregularity. To leverage modern massively parallel processors, e.g. GPUs, makes the problem even harder because of the difficulty of managing massive hardware resources and sophisticated memory hierarchies. In this paper, we investigate the problem of graph coloring which assigns 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 employed in many applications [1–5], and is also intensively utilized by scientific 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 datasets, 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]. There are existing GPU implementations of both catigories. 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, and more importantly they can substantially reduce the total number of memory accesses. But they inherently yield too many colors. On the other hand, SGR implementations [12] generally use fewer colors than MIS ones, but without careful mapping and optimizations, they spend much more time to complete coloring.

To overcome the limitations of existing approaches, we propose a high performance GPU graph coloring implementation which can produce high-quality coloring. Our method is built based on the SGR scheme so that good coloring quality is guaranteed. It is then optimized specifically for the GPU architecture to improve performance. We choose data-driven instead of topology-driven mapping strategy for better work efficincy, and make algorithm tradeoffs to leverage efficient operators and alleviate the side effects of massive parallelism on GPUs. Meanwhile, we incorporate common optimization techniques, e.g. kernel fusion, to further improve performance. The major insight of this work is that *algorithm-specific optimizations* are as important as common optimization techniques for high performance graph algorithm on GPUs. The main contributions of this paper are:

1) We present a work-efficient GPU graph coloring algorithm based on the speculative greedy scheme. The algorithm is carefully refined to better leverage GPU's bulk-synchronous model. It shows the importance of algorithm refinement to achieve high performance on GPUs.

2) We employ optimization techniques specifically for the GPU architecture to take advantage of GPU's computation resources and memory hierarchies. Our practice further demonstrates GPU's capability on accelerating graph algorithms.

3) We implement the proposed algorithm and optimizations using CUDA, and evaluate it on the NVIDIA GPU with both synthetic and real-world sparse 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 serial and parallel algorithms as well

as the state-of-the-art GPU implementations are introduced in Section 2. Our proposed design is presented in Section 3. We present the experimental results in Section 4. Section 5 discusses related work, and Section 6 concludes.

2 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], sparse-matrix computation [6, 7] and assigning frequencies to wireless access points [13].

Graph coloring that minimizes the number of colors is a NP-complete problem, and is known to be NP-hard even solved approximately [14]. 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-offs between minimizing the number of colors and execution time but generally faster algorithms have poor coloring quality while slower ones tend to yield fewer colors. In the following, we introduce some existing sequential and parallel algorithms.

2.1 Sequential Graph Coloring

A sequential algorithm [10, 15] based on the greedy scheme is shown in Algorithm 1. 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 to search 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).

2.2 Parallel Graph Coloring

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

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
```

Gebremedhin and Manne (GM) [9] used *speculation* to deal with the inherent sequentiality of the greedy scheme. It colors as many vertices as possible in parallel, tentatively tolerating potential conflicts, 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 resolve (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 [16].

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 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) [17]. Algorithm 3 shows the details of the JP algorithm. Gjertsen *et al.* [18] 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.* [19] futher improve the ordering heuristics based on the JP algorithm.

2.3 CUDA programming and GPU Graph Coloring

With the success of CUDA [20] programming model, general-purpose graphics processing units (GPGPUs) [21] have been widely used for high performance computing (HPC) and many other application domains during the last decade. In CUDA, individual functions executed on the GPU device are called *kernel* functions, written in a single program multiple-data (SPMD) form. Each instance of the SPMD function is executed by a GPU *thread*. Groups of such threads, called *thread blocks*, are guaranteed to execute concurrently on the same streaming multiprocessors (SMs). Within each group, subgroups of threads called *warps* are executed in lockstep, evaluating one instruction for all threads in the warp at once. One of the major difficulties of CUDA programming is to manage the GPU memory hierarchy. It consists of register files, L1 memories (scratchpad, L1 cache, and read-only data cache), the shared L2 cache, and the off-chip GDDR DRAM [22]. Scratchpad memory (*shared memory* in CUDA terminology) is programmer visible and can be used for explicit intra thread block communication. L2 cache works as the central point of coherency, and is shared across all threads of the entire kernel.

Several GPU graph coloring implementations have been proposed so far using either GM or JP algorithm. Grosset *et al.* [12] implement the GM algorithm using CUDA. 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 assigns as few colors as the serial algorithm, its performance is poor, or even worse than the sequential graph coloring for many datasets, meaning the GPU computation horsepower is not leveraged very well.

The CUSPARSE [23] library offered by NVIDIA includes a `csrcolor` [7] routine which does graph coloring on a given graph in CSR format [24]. The algorithm of `csrcolor` is derived from the JP algorithm, but uses the *multi-hash* method to find 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

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) ← 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
```

can generate $2N$ (maximal) independent sets at once. Compared to the GM algorithm, this method significantly reduces accesses to the *color* array, because it compares the generated hash values (in the registers) instead of the colors of neighbors (in the memory). 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 exploiting 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 K40c 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 is even worse than the sequential implementation, meaning it does 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 better implemention of parallel graph coloring for GPUs to achieve both high performance

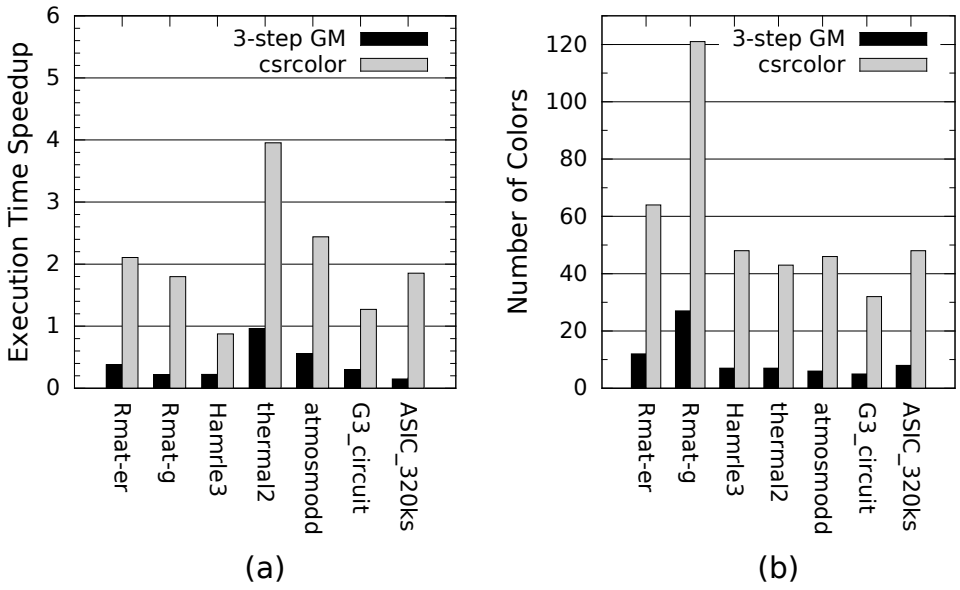

Figure 1: Comparison between two existing GPU graph coloring implementations: `3-step GM` and `csrcolor`. (a) performance, i.e. runtime speedup normalized to the serial implementation (the more the better); (b) coloring quality, i.e. the number of colors assigned (the less the better). This figure shows that existing GPU implementations either have poor performance or yield unsatisfactory coloring quality, which motivates our work.

Algorithm 4 FirstFit routine

```
1: function FIRSTFIT(v)
2:     for each vertex w ∈ adj(v) do
3:         colorMask[color[w]] ← v
4:     end for
5:     c ← min {i > 0 : colorMask[i] ≠ v}
6:     color[v] ← c
7: end function
```

and good coloring quality.

3 Design

Graph algorithms are typical irregular algorithms [25] that are considered to be difficult to parallelize on GPUs. However, recent works [26–31] show that GPUs are capable to substantially accelerate graph algorithms if they are carefully designed and optimized for the GPU architecture. Although the previously proposed optimization techniques for other graph algorithms can be applied to graph coloring, we show that refining the algorithm for GPUs is essential for our case.

As mentioned in Algorithm 2, the graph coloring workload is composed of two major com-

Algorithm 5 ConflictResolve routine

```
1: function ConflictResolve(v)
2:     for each vertex w ∈ adj(v) do
3:         if color[v] = color[w] and v < w then
4:             color[v] ← 0
5:         end if
6:     end for
7: end function
```

ponents: assign the first permissible color (Algorithm 4. `FirstFit`) and resolve conflicting vertices (Algorithm 5. `ConflictResolve`). The operations are trivial, but GPU's massively parallel model makes it challenging to efficiently parallelize these workloads. We investigate the two activities in the following analyses using NVIDIA Tesla K40c GPUs.

Note that we use the well-known compressed sparse row (CSR) [24] 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.

3.1 The Baseline Design

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 baseline algorithm 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 while the CPU is only responsible for controlling the progress. The rationale behind this change of mapping is that throughput-oriented processors are good at exploiting data-level parallelism and thus recomputing the conflicting vertices rather than serializing it onto the CPU would be more straightforward and efficient.

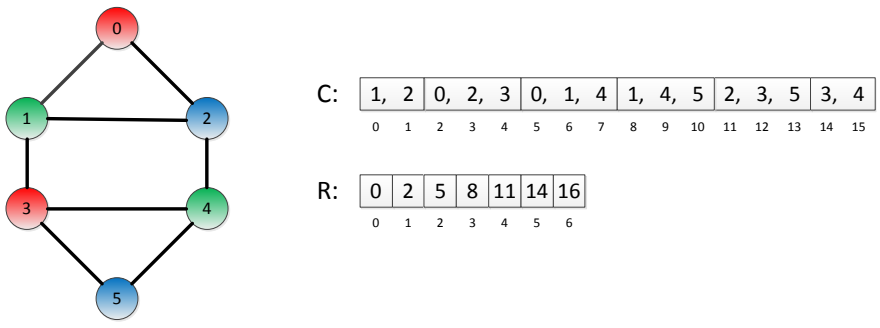

Figure 2: An example of the compressed sparse row (CSR) format. For this graph, at least three colors (red, green, blue) are needed.

Algorithm 6 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 color[v] = 0 then                          ▷ Not colored yet
6:                 FIRSTFIT(v)
7:                 changed ← true
8:             end if
9:         end for
10:        for each vertex v ∈ V in parallel do
11:            if colored[v] = false then                    ▷ Not Colored yet
12:                CONFLICTRESOLVE(v)
13:                if v is not confliting then
14:                    colored[v] = true;
15:                end if
16:            end if
17:        end for
18:    while changed = true
19: end procedure
```

Algorithm 7 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:             FIRSTFIT(v)
6:         end for
7:         W_out ← ∅                                         ▷ Initialize the out worklist
8:         for each vertex v ∈ W_in in parallel do
9:             CONFLICTRESOLVE(v)
10:            if v is confliting then
11:                W_out ← W_out ∪ {v}                       ▷ Atomic push
12:            end if
13:        end for
14:        swap(W_in, W_out)                                 ▷ Swap the worklists
15:    end while
16: end procedure
```

Nasre *et al.* [32] 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 graph coloring in these two fashions. Algorithm 6 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 have been colored, the flag remains $false$ and the algorithm finally terminates. Both `FirstFit` and `ConflictResolve` are similar to those in the GM algorithm, but in `ConflictResolve` a bitmask $colored$ is used to avoid recomputation. Algorithm 7 shows the data-driven graph coloring algorithm. It is almost the same as the GM algorithm except that Algorithm 7 uses *double buffering* [32] 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.

Atomic Operation Reduction. In Algorithm 7, since the out worklist is a shared data structure, pushing elements into the worklist (line 11) 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 [26]. Merrill *et al.* [26] proposed to use software *prefix sum* [33, 34] for updating the shared worklist. Given a list of allocation requirements for each thread, prefix sum computes the offsets for where each thread should start writing its output elements. Fortunately, efficient GPU prefix sums [35] have been proposed, and the CUB [36] library has already provided standard routines for CUDA users to invoke. Thus we need only one atomic operation for each block.

Color Clearing. In Algorithm 7, when a vertex is determined to be conflicting, it is pushed into the worklist. Intuitively, its color should be cleared and it will be assigned color in the next iteraion. However, functionally it is not a necessary operation. In the CPU parallel algorithm, this is not an issue. But for the GPU implementation, it is important to clear the color, so that when its neighbors check its color, there won't be conflicts. Thus in Algorithm 5, the color is cleared (line 4). We observe non-trivial performance drop if the operation is removed. In the following sections we will see that the techniques to alleviate conflicts are performance critical to our GPU implementations.

Fig. 3 compares their performance. As shown in the figure, the data-driven implementation outperforms the topology-driven one on average, although the latter is more intuitive to implement on the GPU. This is easy to understand because the parallelism decreases in graph coloring as the iteration moves forward, and the topology-driven implementation has plenty of threads with no

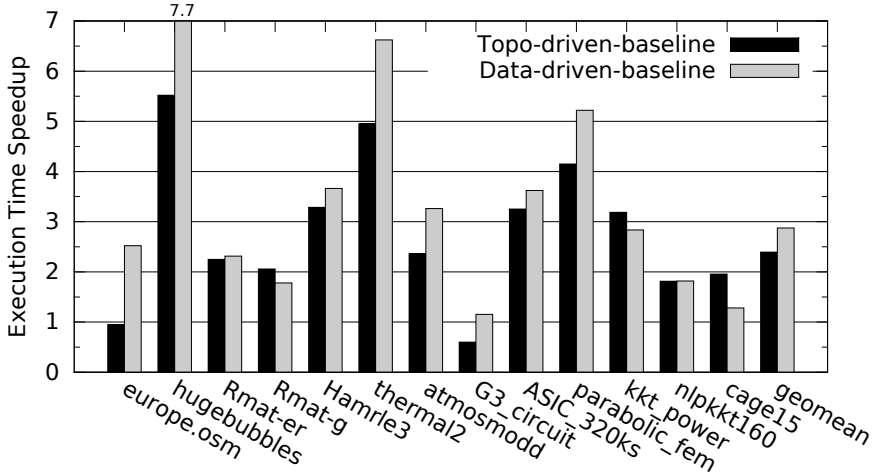

Figure 3: Runtime speedup of topology-driven and data-driven implementations, normalized to the sequential implementation.

work to do, while the data-driven implementation is work-efficient although mantaining the work-list costs extra overhead. In the following discussion, we take this data-driven implementation as our baseline implementation. To achieve higher performance, we refine the algorithm to alleviate the side effects of massive parallelism and leverage efficient operators. We call them *algorithm-specific optimizations*. We also employ common (non-algorithm-specific) optimization techniques in Section 3.3.

3.2 Algorithm Refinement

As most parallel graph processing algorithms, parallel graph coloring is iterative. Therefore it is important to ensure quick convergence for high performance. In the case of graph coloring, the number of iterations required to complete coloring highly depends on the conflict situation. For dense graphs, conflicts happen so frequently that no parallel algorithm can efficiently solve the problem. Our work thus focuses on sparse graph coloring which is more common in real-world applications. Even so it is still challenging to parallelize it on GPUs, because the thousands of threads in the massively parallel programming model make the conflicts happen much more frequently. This is not an issue on CPUs since there are only several or dozens of threads running simultaneously. We propose *heuristic conflict resolve* and employ *thread coarsening* technique to alleviate this side-effect of GPU parallelism.

Heuristic Conflict Resolve. To reduce the number of iterations, an important part is to reduce conflicts. Since conflicts happen when two adjacent vertices are assigned the same color, deciding which of the two conflicting vertices to be re-assigned in the next iteration affects the following conflict situation. An intuitive scheme is to pick the one with smaller or larger vertex id, but this is surely far away from optimal. We apply *heuristic conflict resolve* that prioritizes coloring the vertex

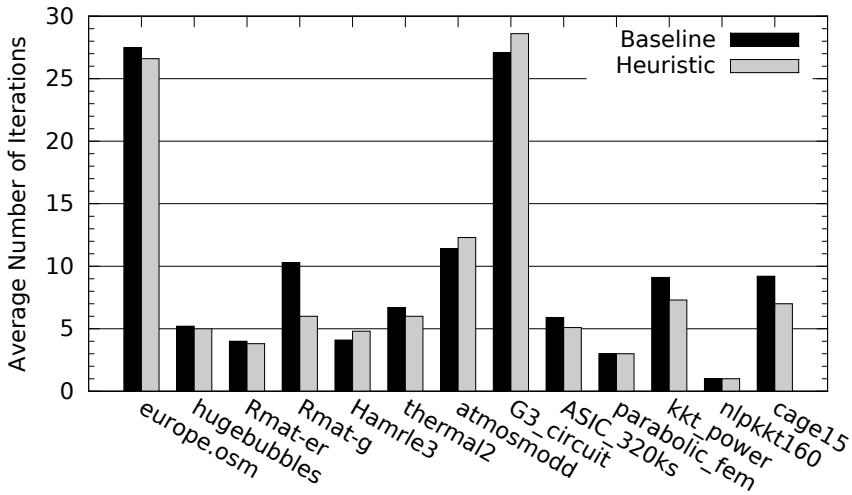

Figure 4: Average number of iterations with the baseline and heuristic implementations. Faster convergence leads to an average of 10.4% entire program speedup over the baseline.

with larger degree and puts the smaller one into the worklist to be processed in the next iteration. The rational behind this heuristic is that vertices with larger degrees have more neighbors and thus are more likely to cause conflicts in the future. So it is better to color large-degree vertices first and reduce the possibility of conflicts. When the two vertices have the same degree, the one with smaller vertex id is picked. Fig. 4 illustrates the average number of iterations required to complete coloring. It is shown that benchmarks e.g. `rmat-g` and `cage15` can have significant iteration reduction using the heuristic. We also observe 43% and 50% execution time speedup of the entire program for the two benchmarks compared to the baseline. On average, the heuristic yields 10.3% speedup over the baseline.

Thread Coarsening. *Thread coarsening* is a common technique utilized in CUDA or OpenCL programs. It merges several threads together and thus have each thread do more work. This reduces the total number of threads and directly affects how data parallel work is mapped to the underlying hardware. Usually it is used to reduce the amount of redundant computation and thus can improve performance. For our case, however, it is used to reduce conflicts, since massive amount of threads on the GPU cause severe conflicts which is not an issue on the CPU. Fig. 5 shows the effect of thread coarsening applied to `FirstFit`, `ConflictResolve` or both. Here we launch $nSM \times max\_blocks$ thread blocks, where nSM is the number of SMs on the GPU and max_blocks is the maximum number of thread blocks that is allowed to be launched on each SM. max_blocks depends on how many resources (e.g. registers, shared memory) a thread block allocates. Each block has 128 threads. Note that this is not the optimal configuration which is different for different benchmarks, and some benchmarks would be faster with even fewer thread blocks. Autotuning techniques would be helpful, but this is out of the range of this paper. As shown, benchmarks e.g. `G3_circuit` and `cage15` can remarkably benefit from thread coarsening. On average, applying thread coarsening on both kernels can improve performance by 4.4% over the baseline.

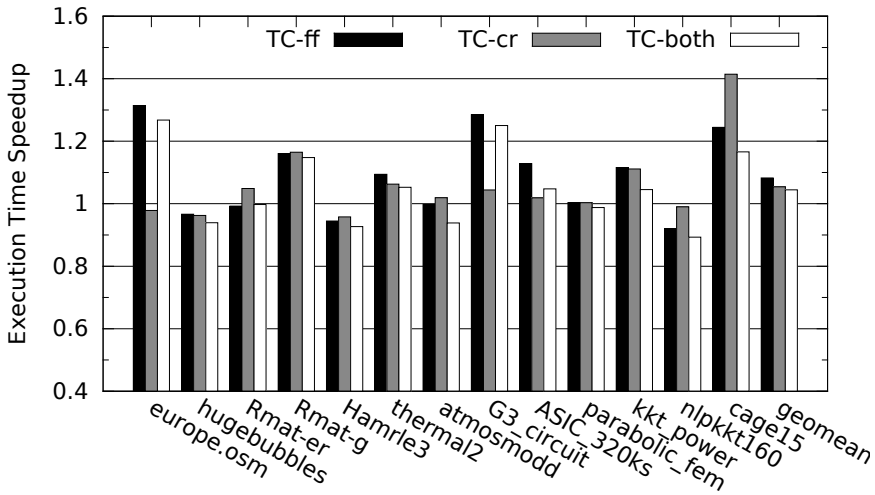

Figure 5: Program execution time speedup of `thread coarsening` on `FirstFit` (TC-ff), `ConflictResolve` (TC-cr) and both kernels (TC-both), all normalized to the baseline.

Bitset Operation. Another major time-consuming part stems from writes and reads on the $colorMask$ data structure in the `FirstFit` kernel. For each vertex, all its neighbors are visited to collect impermissible colors which are written into $colorMask$. This information is then sequentially checked to find the first permissible color. In the worst case, all the elements in the $colorMask$ array are checked, but actually we only need to find one permissble color. To reduce the costs of this operation, we propose to use `bitset` operations to implement reads and writes on the $colorMask$ array. `bitset` is a standard class template in C++, but no similar support is provided in CUDA yet. Thus we implement similar operations to mimic the functionality of the `bitset` class.

Fortunately, NVIDIA GPU architecture provides the `__ffs()` intrinsic for our use. Find first set (ffs) or find first one is a bit operation that identifies the least significant index or position of the bit set to one in the word. So our scheme is to initialize the bits as all "1"s, and clear the bit if the corresponding color is impermissible. To find the first permissible color, we need only to call the `__ffs()` intrinsic. This implemention turns a for-loop into a single instruction and thus significantly reduces the operations required to complete the `FirstFit` kernel. Fig. 6 shows a 61% speedup of the `FirstFit` kernel runtime over the baseline on average when `bitset` is applied. We also observe that this kernel improvement leads to an average of 28% speedup of the entire program compared to the baseline.

3.3 Common Optimization Techniques

Existing GPU graph processing algorithms have already utilized many optimization techniques to improve performance. In graph coloring we employ some of these optimizations, including *kernel fusion*, *read-only data caching*, and *load balancing* to enhance our implementation.

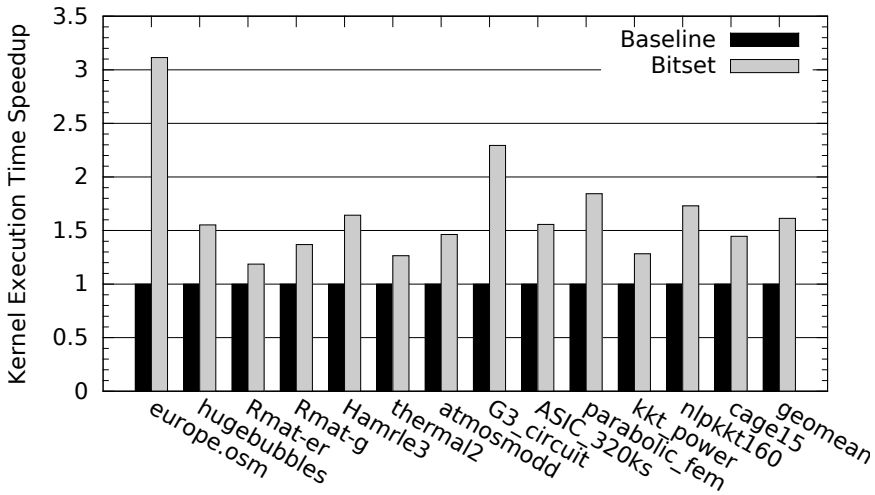

Figure 6: `FirstFit` kernel execution time speedup of `bitset` over the baseline. The kernel execution time is obtained by `nvprof`.

Kernel Fusion. Previous techniques focus on individual kernels. However, another important optimization technique called *kernel fusion* combines multiple GPU kernels into a single one, and thus can keep the entire program on the GPU. Since adjacent kernels in CUDA share no state, this technique can leverage producer-consumer locality between operations and thus save significant memory bandwidth [37]. Note that global barrier is required between `FirstFit` and `ConflictResolve` operations. We use the existing method proposed by Xiao *et al.* [38]. With this global barrier, kernels can only launch limited number of thread blocks, and thus thread coarsening is forced to be applied to both kernels. Fig. 7 shows an average 10% speedup of kernel fusion over the baseline. As shown, benchmarks e.g. `cage15` and `rmat-g` can benefit from better locality brought by kerel fusion since they are relatively denser and more irregular than others. We observe an improved L2 cache hit rate for `cage15`.

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 be kept in the read-only data (unified L1/texture) cache by reading it using the intrinsic `__ldg()` [20]. We use the texture cache to hold the read-only data, i.e. the C array and the R array. And then more read-only data is forced to 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 capture temporal locality and improve the performance because of reduced DRAM accesses. As shown in Fig. 7, `__ldg()` can bring 3.6% speedup over the baseline.

Load Balancing. Another important issue for graph algorithms is load imbalance. The problem is particularly worse for scale-free (power-law) graphs. Merrill *et al.* [26] proposed a hierarchical load balancing strategy which maps the workload of a single vertex to a thread, a warp, or a thread block, according to the size of its neighbor list. At the fine-grained level, all the neighbor list offsets in the same thread block are loaded into shared memory, then the threads in the

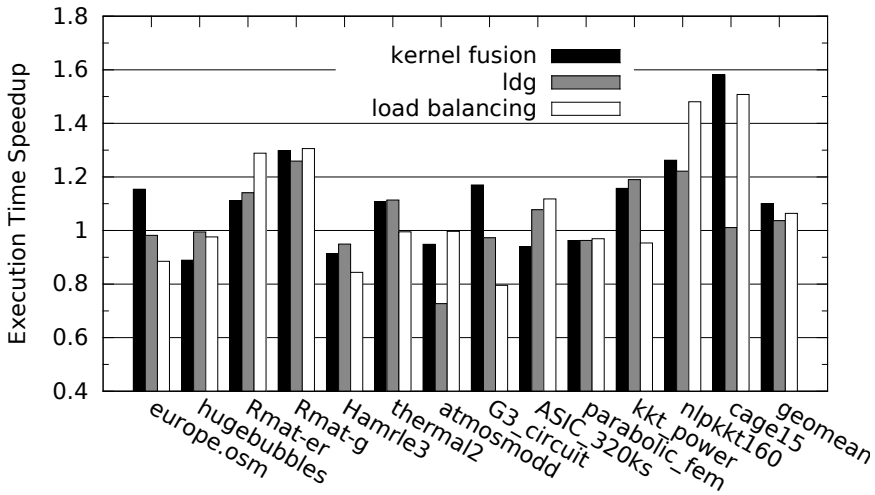

Figure 7: Program execution time speedup of *kernel fusion*, *ldg* and *load balancing* over the baseline.

block cooperatively process per-edge operations iteratively. At the coarse-grained level, per-block and per-warp schemes are utilized to handle the extreme cases: (1) neighbor lists larger than a thread block; (2) neighbor lists larger than a warp but smaller than a thread block respectively. We implement this strategy on graph coloring. Fig. 7 illustrates the effect of load balancing on the benchmarks. Irregular benchmarks with uneven degree distribution, e.g. `rmat-g` and `cage15` can substantially benefit from this technique. On average, it achieves 6.4% speedup over the baseline.

4 Evaluation

We use the R-MAT [39] graph generator to create synthetic graphs. The R-MAT algorithm determines the degree 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 real-world sparse graphs from the University of Florida Sparse Matrix Collection [40]. These benchmarks are also used in previous work [7,26]. The matrices with the respective number of vertices (i.e. rows) and edges (non-zero elements) are shown in Table 1. The graphs vary widely in size, degree distribution, density of local subgraphs and application domain.

4.1 Experiment Setup

We compare 6 implementations including (1) `Serial`: the serial implementation in CUSP [15], (2) `OpenMP`: the baseline OpenMP implementation in [10], (3) `3-step GM`: the previously proposed GM GPU implementation [12], (4) `csrcolor`: the routine provided by NVIDIA CUS-

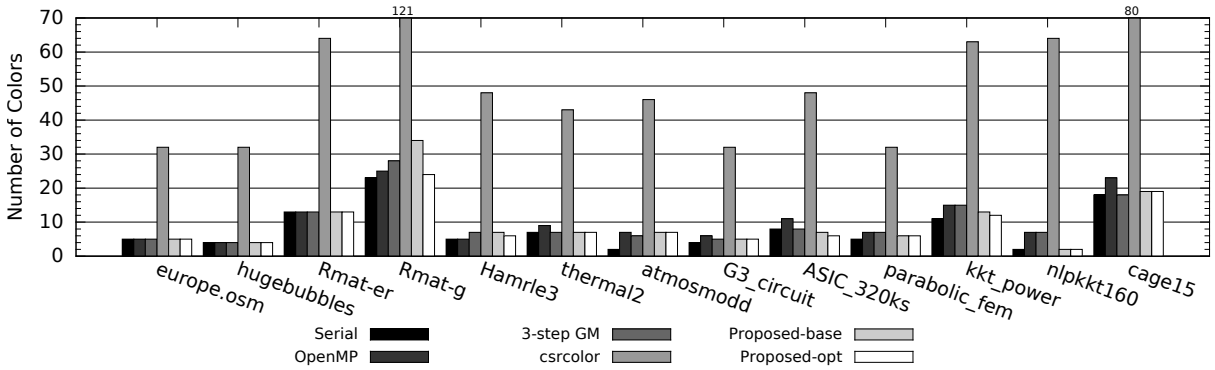

Figure 8: Total number of colors assigned with different implementations.

PARSE [7], (5) `Proposed-base`: our proposed baseline data-driven implementation, (6) `Proposed-opt`: our proposed optimized data-driven implementation. We conduct the experiments on the NVIDIA K40c GPU with CUDA Toolkit 7.5 release. `Serial` is executed on Intel Xeon E5 2670 2.60 GHz CPU with 12 cores. All the benchmarks are executed 10 times and we collect the average execution time to avoid system noise. Timing is only performed on the computation part of each program. For all the GPU implementations, the input/output data transfer time (usually takes 10%-20% of the entire program execution time) is excluded because data is resident on the GPU in real applications [7].

Name	$n(10^6)$	$m(10^6)$	$\bar{d}$	σ	**Description**
`europe.osm`	50.9	108.1	2.1	0.23	Road Network
`hugebubbles`	21.2	63.6	3.0	0	Adaptive Mesh
`rmat-er`	1.0	10.0	10.0	10.83	Synthetic
`rmat-g`	1.0	10.0	10.0	123.34	Synthetic
`Hamrle3`	1.4	11.0	7.6	7.2	Circuit Sim.
`thermal2`	1.2	8.6	7.0	0.7	Thermal Sim.
`atmosmodd`	1.3	8.8	6.9	0.1	Atmosphere
`G3_circuit`	1.6	7.7	4.8	0.4	Circuit Sim.
`ASIC_320ks`	0.3	1.8	5.7	63.2	Circuit Sim.
`parabolic_fem`	0.5	3.7	7.0	0.02	General
`kkt_power`	2.1	14.6	7.1	54.8	Optimization
`nlpkkt160`	8.3	229.5	27.5	7.3	Optimization
`cage15`	5.2	99.2	19.2	32.9	Electrophoresis

Table 1: Suite of benchmark graphs. n: number of vertices, m: number of edges, $\bar{d}$: average dergee, σ: degree variance.

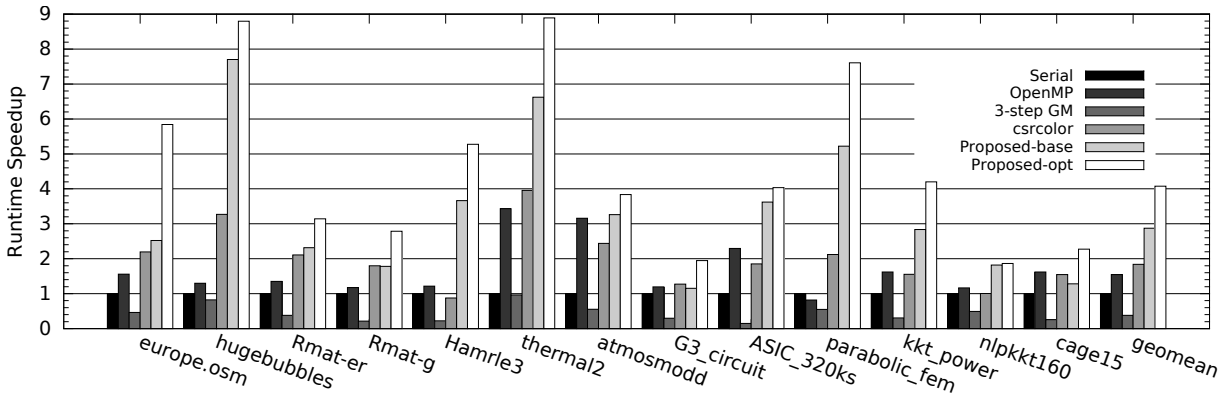

Figure 9: Runtime speedup normalized to the serial algorithm.

4.2 Coloring Quality

Fig. 8 shows the number of colors needed by different implementations for each graph. It is not surprising that implementations except `csrcolor` need similar amount of colors, since they are all based on the greedy scheme. The slight difference among these 5 implementations may result from the different orderings that are caused by different thread mapping stratigies and so on. `csrcolor`, however, needs 3.9×∼31× more colors than `Serial`, making this MIS based implementation unattractive or even unapplicable in many scenarios. This substantial difference of coloring quality between `csrcolor` and the other implementations stems from the inherent algorithm property of the SGR scheme and the MIS scheme. SGR uses greedy scheme, and for parallel versions it optimistically does coloring in parallel with later conflict resolve. MIS, however, 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.

4.3 Performance

Fig. 9 illustrates the execution time speedup normalized to `Serial`. `OpenMP` on CPU achieves only moderate speedup (1.54×). As mentioned before, `3-step GM` gets unacceptable performance: 62% average slowdown compared to `Serial`. The slowdown stems from its mapping strategy and different data representation. In contrast, `csrcolor` is a much faster GPU implementation. It achieves an average speedup of 1.84× over `Serial`. For regular graphs, such as `hugebubbles` and `parabolic_fem`, it performs much better than `OpenMP`. This shows the high throughput and bandwidth advantages of GPUs over CPUs.

Our proposed baseline implementation performs even better than `csrcolor`. We observe 2.87× speedup on average over `Serial`. It is 85.8% and 56.1% faster than `OpenMP` and `csrcolor` respectively. For some benchmarks e.g. `Hamrle3` and `parabolic_fem`, `Proposed-base` significantly outperforms `csrcolor` (4.18× and 2.46×). This performance boost mainly comes from the selection of data-driven algorithm structure and the atomic operation reduction. However,

for relatively dense or irregular benchmarks, e.g. `cage15`, it performs worse than `csrcolor`, because no specific work is done to handle irregular cases and `csrcolor` has fewer memory accesses as mentioned before.

With careful algorithm refinement and optimization techniques, we further improve the performance with an average speedup of 4.08× over `Serial`. It is 2.63×, 2.21× and 1.42× speedup over `OpenMP`, `csrcolor` and `Proposed-base` respectively. Generally, for regular benchmarks, it takes advantage of GPU's high throughput as `csrcolor` does, and performs even better because of the efficient bitset operator, fast convergence and so on, e.g. `hugebubbles` (8.8×) and `thermal2` (8.9×). For irregular benchmarks, better locality and load balance lead to better performance. Thus `Proposed-opt` can consistently outperform existing CPU and GPU parallel implementations.

We also notice that for some benchmarks, e.g. `G3_circuit` and `nlpkkt160`, `Proposed-opt` gets very limited performance improvement compared to `Serial`. It is clear that the performance of graph coloring highly depends on the graph characteristics (scale, density, degree distribution and topology). For example, `nlpkkt160` has a relatively large average degree and suffers from conlicts. And some are small in size, which limits the potential of performance improvement using GPUs. But more importantly, since the compute operation is trivial, the performance is likely to be limited by memory operations. For sparse graphs, not much temporal locality exists, and thus the kernel becomes extremely memory bound with large-scale datasets, which could not be mitigated by the optimizations that we employ. We suggest system software or hardware support for efficient memory access to overcome this performance bottleneck.

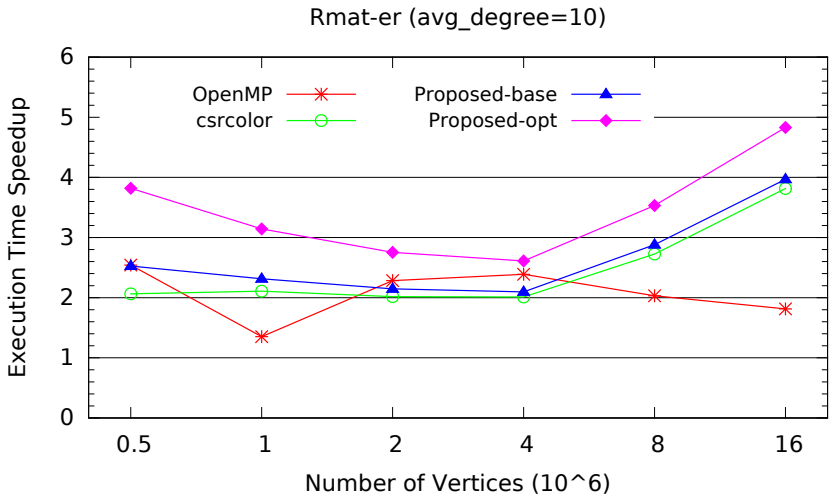

Figure 10: Execution time speedup of `Rmat-er` and `Rmat-g` with various graph size (number of vertices), all normalized to `Serial`.

4.4 Scalability

To evaluate the scalability of our design on the input size, we vary the graph size (number of vertices) of `Rmat-er` from 500K to 16M with fixed average degree ($\bar{d} = 10$). Fig. 10 illustrates that `Proposed-opt` could achieve even more performance speedup given larger input datasets. Our proposed implementation can consistently gain more than 2.5× speedup as the graph size changes, and always outperforms `csrcolor`. After 4M vertices, the speedup increases siginificantly as the graph size increases, while `OpenMP` changes moderately and even drops at the extremely large size. There is also a slight drop around 3M size for GPU impelementaions. This drop is related to the graph characteristics on which the performance highly depends on as mentioned. Here the cause is most likely the graph topology and degree distribution. Even so, `Proposed-opt` is still 9.3% faster than `OpenMP` at the 4M size, while it gets 2.66 speedup over `OpenMP` at the 16M size. For `Rmat-g` (not illustrated), we see a similar trend.

4.5 Sensitivity to Density

As mentioned, graph algorithms are highly sensitive to the characteristics of the input datasets. We evaluate sensitivity to the graph density of our proposed graph coloring implementation. In Fig. 11, we vary the average degree $\bar{d}$ of `Rmat-er` with fixed graph size (1M vertices). We compare `OpenMP`, `csrcolor`, `Proposed-base` and `Proposed-opt`, all normalized to `Serial`. As shown, `Proposed-opt` significantly outperforms the others when $\bar{d}$ is small. This means our proposal can efficiently handle sparse graphs. However, as the average degree increases, the performance improvement over `Serial` decreases for `csrcolor` and our proposals. Their curves drop blow `OpenMP` when $\bar{d}$ is larger than 20.

In contrast, `OpenMP` is more stable than GPU implementations. The drop of GPU ones results from the conflicts between neighbors which is not an issue in CPUs. As the graph becomes denser, the conflicts happen more frequently. In this case, the GPU implementations need much more iterations to complete than `OpenMP`. For dense graphs, thanks to the techniques that alleviate conflicts, `Proposed-opt` still achieves comparable performance to `csrcolor`, while `Proposed-base` becomes worse than the other two GPU ones and finally becomes slower than the serial implementation (blow 1). Remeber that our proposals still consistently yield much better coloring quality than `csrcolor`. Although GPU implementations achieves high performance for sparse graphs, for dense graphs we suggest to use CPUs instead of GPUs to solve the graph coloring problem.

5 Related Work

Plenty of previous works have investigated parallelizing graph coloring [10, 16, 18, 19]. There are two major classes of parallel graph coloring algorithms: the GM algorithm [9] proposed by Gebremedhin and Manne and the JP algorithm [11, 17] proposed by Jones and Plassmann. Grosset *et al.* [12] implement `3-step GM` on GPUs based on the GM algorithm, while the `csrcolor` [7] routine in CUSPARSE [23] uses the JP algorithm. Our proposed implementation is

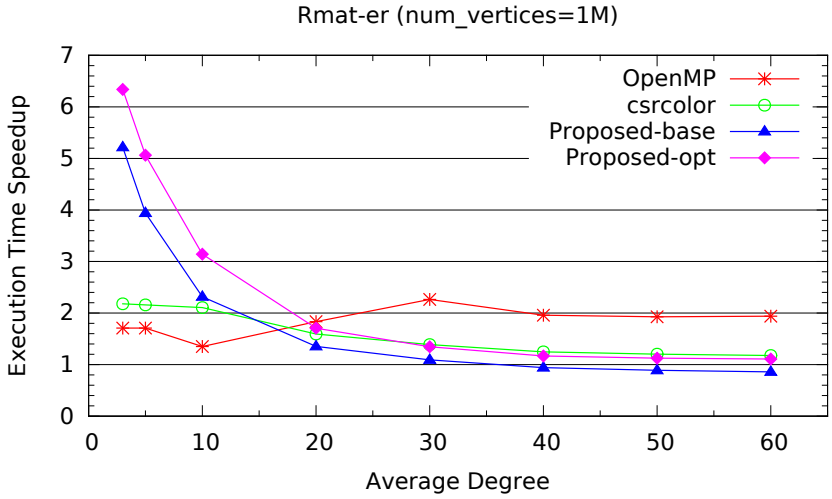

Figure 11: Execution time speedup of `Rmat-er` with various average degrees, all normalized to `Serial`.

based on GM algorithm, but achieves much better performance than `3-step GM` with algorithm refinement and architecture-oriented optimizations. It also outperforms `csrcolor`, and produces much better coloring quality because of the algorithm inherent property.

Many other graph algorithms have been developed on GPUs. Harish *et al.* [41] are the pioneers to implement GPU graph algorithms. They developed topology-driven Breadth-first Search (BFS) and shortest path algorithms. Hong *et al.* [42] proposed another topology-driven BFS to map warps rather than threads to vertices. Luo *et al.* [43] developed the first work-efficient BFS on GPUs. Merrill *et al.* [26] improved Luo's work. They employed prefix sum to reduce atomic operations and used dynamic load balancing to deal with scale-free graphs. This implementation thus achieves high throughput and good scalability. The two major techniques of their work are also applicable to our implementation, while our work focuses more on the algorithm-specific refinement, e.g. the specific strategies to alleviate side effects of GPU's massive parallelism.

Davidson *et al.* [31] developed a work-efficient Single-Source Shortest Path (SSSP) algorithm on the GPU. They used another load balancing strategy which partitions the work into chunks and assigns each chunk to a block. Reserchers also proposed GPU implementations of Betweenness Centrality [27], Minimum Spanning Tree [30,44], Strongly Connected Components [28] and so on. These work together demonstrated that with careful mapping and optimizations graph algorithms can get substantial performance boost on the GPU. Our work further enhances the conclusion of previous practices, while we show the importance of algorithm refinement and architecture-specific optimizations for the problem of graph coloring.

Researchers have proposed many optimization techniques for graph algorithms, or more generally, for irregular algorithms on GPUs. LAVER [45] is a locality-aware vertex scheduling scheme which reorders the vertex queue to improve temporal locality of vertex data stored in on-chip caches. Nasre [46] proposed high-level methods to eliminate atomics in irregular programs, e.g. BFS and SSSP, on GPUs. Gunrock [37] absorbs previous knowledge and provides a library so-

lution for GPU graph processing. It provides a load balancing framework based on Merrill's and Davidson's strategies, and integrates a set of common optimization techniques. A huge amount of efforts [47–54] have been made by researchers to generalize graph processing computation and reduce programmer's burden. Although generalized method can improve programmability, we argue that optimizations customized for the specific algorithm (which is difficult to generalize) is also important.

Che *et al.* [55] characterize a suite of GPU graph applications and suggest architectural support. Xu *et al.* [56] evaluate existing GPU graph algorithms on both a GPU simulator and a real GPU card and also suggest GPU hardware support. Wu *et al.* [57] characterize three GPU graph frameworks and suggest to focus on constructing efficient operators. Beamer *et al.* [58] also measure three graph libraries and propose processor architecture change. Green-Marl [59] is a domain specific language for graph processing. Chen *et al.* [60] proposed compiler optimization methodology for graph and other irregular applications on Intel Xeon Phi coprocessors. Ahn *et al.* [61] developed a customized Processing-in-Memory (PIM) accelerator for large-scale graph processing. We believe that language, compiler, runtime and architecture support is necessary for large-scale graph processing.

6 Conclusion

Graph coloring is an important graph algorithm that has been applied in many application domains. To process large-scale graphs, parallel graph coloring has been intensively studied in the past. Meanwhile, GPUs have been broadly utilized to speed up compute intensive kernels of HPC applications in the past decade. In this paper, we explore parallel graph coloring on the GPU. Existing implementations either achieve limited performance or yield unsatisfatory coloring quality. We present a high performance graph coloring implementation for GPUs with good coloring quality. We utilize the speculative greedy scheme that guarantees coloring quality, and improve performance with algorithm refinement and common optimization techniques. Experimental results show that our proposed implementation outperforms existing GPU implementations in terms of both performance and coloring quality. This work helps us further understand graph algorithms on modern massively parallel processors and gives insight on the importance of both algorithm-specific and non-algorithm-specific (common) optimizations. We also show the necessity of lower level support from system software and architecture.

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 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 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 multi-core 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 17th Annual ACM Symposium on Theory of Computing*, pp. 1–10, 1985.

[12] 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.

[13] 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.

[14] D. Zuckerman, "Linear degree extractors and the inapproximability of max clique and chromatic number," in *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pp. 681–690, 2006.

[15] 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/

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sept 2011.

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

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

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

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 349–359, May 2014.

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

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

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

[35] 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.

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

[37] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2016.

[38] S. Xiao and W. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Proceedings of the IEEE 24th International Parallel and Distributed Processing Symposium*, pp. 1–12, May 2010.

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

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

[41] P. Harish and P. J. Narayanan, *Proceedings of the 14th International Conference High Performance Computing (HiPC)*. Berlin, Heidelberg: Springer Berlin Heidelberg, December 2007, ch. Accelerating Large Graph Algorithms on the GPU Using CUDA, pp. 197–208.

[42] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pp. 267–276, 2011.

[43] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, pp. 52–55, 2010.

[44] V. Vineet, P. Harish, S. Patidar, and P. Narayanan, "Fast minimum spanning tree for large graphs on the gpu," in *Proceedings of the Conference on High Performance Graphics*, p. 167171, 2009.

[45] H. Park, J. Ahn, E. Park, and S. Yoo, "Locality-aware vertex scheduling for gpu-based graph computation," in *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 195–200, Oct 2015.

[46] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on gpus," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pp. 96–107, 2013.

[47] 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 ACM SIGMOD International Conference on Management of Data*, pp. 135–146, 2010.

[48] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pp. 456–471, 2013.

[49] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new parallel framework for machine learning," in *Proceedings of the UAI*, pp. 340–349, 2010.

[50] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pp. 17–30, 2012.

[51] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 135–146, 2013.

[52] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, June 2014.

[53] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: Vertex-centric graph processing on gpus," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pp. 239–252, 2014.

[54] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, pp. 2:1–2:6, 2014.

[55] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pp. 185–195, Sept 2013.

[56] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?" in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pp. 140–149, Oct 2014.

[57] Y. Wu, Y. Wang, Y. Pan, C. Yang, and J. D. Owens, "Performance characterization of high-level programming models for gpu graph analytics," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pp. 66–75, Oct 2015.

[58] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pp. 56–65, Oct 2015.

[59] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: A dsl for easy and efficient graph analysis," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 349–362, 2012.

[60] L. Chen, P. Jiang, and G. Agrawal, "Exploiting recent simd architectural advances for irregular applications," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 47–58, 2016.

[61] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pp. 105–117, 2015.