Evaluating Graph Coloring on GPUs

Name1 Name2 Name3 Name4 Name5

Affiliation

Email1/2/3/4/5

Abstract

This paper evaluates features of graph coloring algorithms implemented on graphics processing units (GPUs), comparing coloring heuristics and thread decompositions. As compared to prior work on graph coloring for other parallel architectures, we find that the large number of cores and relatively high global memory bandwidth of a GPU lead to different strategies for the parallel implementation. Specifically, we find that a simple uniform block partitioning is very effective on GPUs and our parallel coloring heuristics lead to the same or fewer colors than prior approaches for distributed-memory cluster architecture. Our algorithm resolves many coloring conflicts across partitioned blocks on the GPU by iterating through the coloring process, before returning to the CPU to resolve remaining conflicts. With this approach we get as few color (if not fewer) than the best sequential graph coloring algorithm and we are not too far off the fastest sequential graph coloring algorithms which had poor color quality.

Keywords Graph coloring, Parallel algorithm, GPU, CUDA

1. Introduction

Graph coloring refers to the assignment of labels or colors to elements of a graph (vertices or edges) subject to certain constraints. In this paper, we consider the specific problem of assigning colors to vertices so that no two neighboring vertices (vertices connected by an edge) have the same color. There are several known applications of graph coloring like assigning frequencies to wireless access points, time-tabling and scheduling, register allocation and printed circuit testing among others.

Graph coloring is NP-hard, and even computing a $n^{1-\epsilon}$-approximation of the chromatic number of a graph is NP-hard [5]. Therefore, a number of heuristics have been developed to assign colors to vertices; some commonly used heuristics include First Fit, Largest Degree Order and Saturation Degree Order[2]. These heuristics tend to trade off minimizing the number of colors and minimizing execution time but generally the faster algorithms have poor coloring quality while the slow ones tend to be better. Combinations of these algorithms have also been used to create better heuristics like the combined SDO & LDO [2].

This paper examines a graphics processing unit (GPU) mapping of parallel graph coloring. Prior parallel graph coloring algorithms have been evaluated on conventional shared-memory multiprocessors [4] or distributed systems [3] but to the best of our knowledge, this is the first study of how GPU architectures affect performance gains and number of assigned colors in a parallel implementation. Our study demonstrates that features of the GPU architecture significantly impact the algorithms selected. Specifically, the support for efficient fine-grain multithreading facilitates strong performance gains over CPU implementations. Because hundreds or even thousands of threads can be applied to the parallel coloring, we can obtain as few colors as the best sequential algorithm while operating nearly as fast as the fastest sequential algorithms.

GPUs have different architectures compared to parallel computers where most of the parallel graph-coloring algorithms are run. The main differences is that they have many processors, for example, a Tesla S1070 processor has 240 cores, but each of the cores is slower than a state-of-the-art CPU. Also, a GPU stores the data in the global memory which all the cores can readily access.

2. Algorithm

In this paper, we propose a framework based on the G-M algorithm [4], which is adapted for an NVIDIA GPU platform. In addition, two new heuristics are proposed to match the parallelism exploited by the GPU, which are shown to give better coloring quality than FF and the same or better quality as SDO with the running time that is comparable to FF.

2.1 The Graph Coloring Framework

Algorithm 1 Graph Coloring Framework

Phase 1 : $Graph\ Partitioning - CPU$
  Logically partition graph into subgraphs
  Identify boundary nodes
  Count neighbors outside the subgraph for each vertex

Phase 2 : $Graph\ Coloring\&Conflict\ Solving\ - GPU$
  while Number of color conflicts is high **do**
    Color graph using the specified heuristic
    Identify color conflicts

Phase 3 : $Sequential\ Conflicts\ Rsolution\ - CPU$
  Residual conflicts are eliminated

Phase 1: Graph Partitioning: The graph is initially padded with empty nodes so that it is exactly divisible by the total number of threads. Each thread equally takes N/p vertices, which is a load-balanced approach.

Phase 2: Graph Coloring & **Conflicts detection:** Each thread assigns colors to its subgraph but checks the whole graph before allocating colors. We tried four heuristics including two proposed heuristics (MAX OUT and MIN OUT)for color allocation:

i) First Fit: Allocate the smallest possible color to each vertex

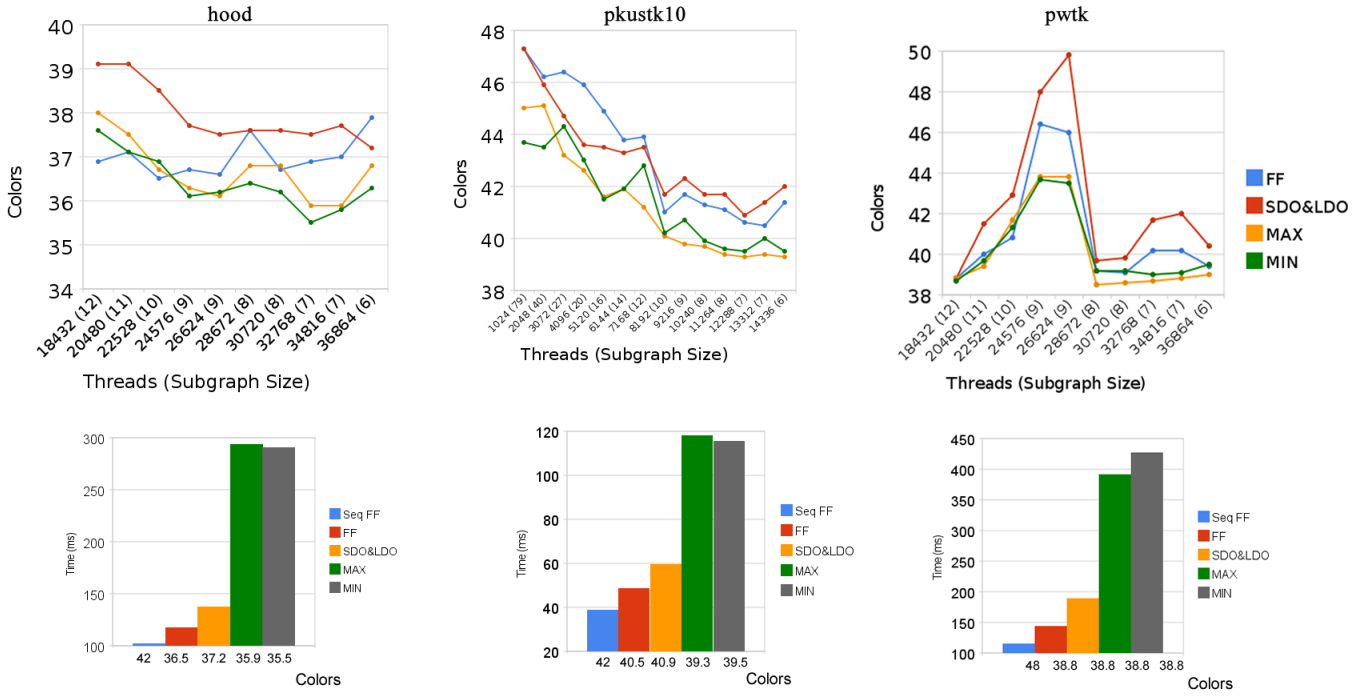

Figure 1. First row: Colors vs Threads (Subgraph size) Second row: best performance for each algorithm

ii) SDO & LDO: Color is allocated to the vertex having the highest saturation and then highest degree

iii) MAX OUT: Color is allocated to the vertex having most neighbors out of the subgraph and then highest degree

iv) MIN OUT: Color is allocated to the vertex having fewest neighbors out of the subgraph and then highest degree

Given that coloring is being performed in parallel, conflicts occur between vertices found in different subgraphs. Conflicts are identified by assigning one thread per boundary node which checks for color conflicts and reset the color of these nodes to null.

Phase 3: Conflicts Resolving Dropping to the CPU to solve the conflicts is common in many approaches but given that we have many small partitions, we tend to have lots of conflicts and resolving that on the CPU can be quite slow. So we do multiple passes of step 2 on GPU until the number of conflicts become very small and can be quickly solved sequentially on the CPU.

3. Experiments and Results

The algorithm has been implemented using the CUDA API and the tests were carried out on a Tesla S1070 with real graphs from the University of Florida Sparse Matrix Collection [1]. In this paper we will focus on 3 different graphs shown in Table 1

				Colors		
Name	n	m	Δ	FF	SDO & LDO	FF Time
hood	220,542	5,273,947	76	42	36	104.7
pkustk10	81,920	2,114,154	89	42	42	39
pwtk	217,918	5,926,171	179	57	48	48

Table 1. Graph properties - n: number of vertices, m the number of edges & Δ: maximum vertex degree

To investigate the impact of parallel graph coloring algorithms on both performance and number of colors obtained, the number of threads is linearly incremented and the timing and color obtained is compared and the results are shown in Figure 1

The results show how different algorithms affect the number of colors and performance as a function of the number of threads. MAX and MIN generally yield fewer colors than any other algorithm and we also notice that good coloration is obtained at relatively small graph sizes. In terms of speed among the parallel graph coloring algorithms, First Fit is the fastest followed by SDO & LDO and MIN and MAX which take the same amount of time.

4. Conclusion

Though our GPU implementation is slower than the reported fastest Sequential Graph Coloring algorithm, in terms of color quality, we are much better; we sometimes even beat the sequential SDO & LDO as well as colors reported in [3].

References

[1] The university of florida sparse matrix collection. URL `http://www.cise.ufl.edu/research/sparse/matrices/`.

[2] H. Al-Omari and K. Sabri. New graph coloring algorithms. *Am. J. Math. & Stat*, 2(4):739–741, 2006.

[3] A. M. F. B. E. C. Bozdag, D Gebremedhin. A framework for scalable greedy coloring on distributed-memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535, 2008.

[4] A. Gebremedhin and F. Manne. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, 12(12):1131–1146, 2000.

[5] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3(1):103–128, 2007. doi: 10.4086/toc.2007.v003a006. URL `http://www.theoryofcomputing.org/articles/v003a006`.