Parallel Graph Coloring in CUDA: A Comparative Analysis

Report of the project for the GPU Computing course held by Professor Giuliano Grossi, University of Milan.

Academic Year
2022-2023

Giorgio Sepe

giorgio.sepe@studenti.unimi.it

Abstract

This project report presents an exploration of parallel graph coloring techniques implemented in CUDA (Compute Unified Device Architecture) follow the Jones-Plassmann approach. Graph coloring is a fundamental problem with applications in resource allocation, scheduling, and optimization, among others. To assess the performance of the implemented algorithms, a set of benchmark graphs is employed, representing a diverse range of graph structures and sizes. The report includes a comparative analysis of the runtime and coloring quality achieved by different heuristics.

Contents

1 Introduction **2**

2 Design and Implementation **2**

2.1 Input Data and Graph Representation 2

2.2 The Sequential Algorithm 2

2.3 Parallel Algorithms 3

2.4 CUDA Implementation 4

2.4.1 Initialization 4

2.4.2 Coloring 4

2.5 Random Priority Heuristic 4

2.6 Largest-Degree-First ordering heuristic 4

2.7 Smallest-Degree-Last ordering heuristic 4

2.8 Saturation Ordering Heuristic 5

2.9 Incidence-Degree Ordering Heuristic 5

3 Results and Conclusions **5**

3.1 Devices and Code Comparison 5

3.2 Input Graphs 5

3.3 Results 6

3.4 Conclusions 8

1 Introduction

The graph coloring problem is a classic and widely studied computational problem that involves assigning colors to the vertices of a graph in such a way that no two adjacent vertices share the same color. The primary objective is to minimize the number of colors used, known as the chromatic number while ensuring that adjacent vertices have distinct colors, the chromatic number problem is an NP-complete problem. Some key areas where graph coloring plays a crucial role include: Scheduling and Resource Allocation, Register Allocation in Compilers, Wireless Network Channel Assignment, Map Coloring in Geography, VLSI Design.

The subsequent section provides an overview of the used algorithms and the implementation in CUDA will be discussed, Following that, performance metrics and benchmarks are then introduced to assess the efficiency and scalability of the CUDA implementations.

2 Design and Implementation

2.1 Input Data and Graph Representation

The data structure used to represent a graph is defined in the class `Graph` of the project. The graph is defined using the "adjacency list" because is particularly efficient for sparse graphs where the number of edges is much smaller than the total possible number of edges. In the internal state, there is the `GraphStruct` that contains:

1. `neighs`: the adjacency list for all nodes.
2. `neighIndex`: the index to find the adjacency list of each node in the full adjacency list array mentioned above.

Here there is a brief explanation of the main methods of the class:

1. `readFromMtxFile(const char* mtx)`: permits to reading a graph represented in the mtx format.
2. `getDeviceStruct(GraphStruct*& dest)`: alloc and copy the graph in the device memory.

The results are evaluated using different public datasets of graphs with different characteristic and they are all listed in the references.

2.2 The Sequential Algorithm

Sequential algorithms are not the main focus of the project, therefore, a greedy algorithm is implemented as a reference. A greedy algorithm is a heuristic algorithm that makes locally optimal choices at each step with the hope of finding a globally optimal solution.

Algorithm 1 is implemented in the `SequentialGreedyColorer::color` method. It takes a graph as input and returns a Coloring struct containing the assigned colors.

- The algorithm starts by initializing data structures, such as arrays to store colors (coloring) and flags indicating whether a vertex is colored (coloredNodes).
- The first vertex is assigned a color, and then the algorithm iteratively colors the remaining vertices using a greedy approach. It considers the colors of neighbouring vertices and chooses the first available color.
- The result is stored in a Coloring struct.

Algorithm 1 Sequential Greedy Coloring Algorithm

```
1: $n \leftarrow |V|$
2: Choose a random permutation $p(1), \dots, p(n)$ of numbers $1, \dots, n$
3: $U \leftarrow V$
4: for $i \leftarrow 1$ to $n$ do
5:     $v \leftarrow p(i)$
6:     $S \leftarrow$ colors of all colored neighbors of $v$
7:     $c(v) \leftarrow$ smallest color not in $S$
8:     $U \leftarrow U \setminus \{v\}$
9: end for
```

Please note that this is a simple implementation, and there may be room for optimizations based on the specific characteristics of the graph.

2.3 Parallel Algorithms

The implemented algorithms are based on the Luby's parallel maximal independent set (MIS) algorithm and the improved version formalized by Jones-Plassman. The maximal independent set is a set of vertices in a graph such that no two vertices in the set are adjacent. A pseudo-code of the JonesPlassmann algorithm is given in algorithm 2.

Algorithm 2 The parallel Jones-Plassmann algorithm for coloring a graph

```
1: $U \leftarrow V$
2: while $|U| > 0$ do
3:     for all $v \in U$ in parallel do
4:         $I \leftarrow \{u \mid w(v) > w(u)$, for all neighbors $u \in U\}$
5:         for all $v_0 \in I$ in parallel do
6:             $S \leftarrow$ colors of all neighbors of $v_0$
7:             $c(v_0) \leftarrow$ minimum color not in $S$
8:         end for
9:     end for
10:    $U \leftarrow U \setminus I$
11: end while
```

The following ordering heuristics are implemented in the project:

- random ordering (R): where the vertices are colored in random order.
- largest-degree-first ordering (LDF), where the vertices with larger degrees are colored first.
- smallest-degree-last ordering (SDL), where the vertices with the smallest degree are successively removed from the graph, the modified graph is colored using the LDF heuristic, and finally the removed vertices are gradually re-inserted and colored.
- saturation-degree ordering (SD), where the vertices whose colored neighbours have the largest number of unique colors are colored first.
- incidence-degree ordering (ID), where the vertices with the largest number of colored neighbors are colored first irrespective of the number of unique colors.

2.4 CUDA Implementation

This implementation consists of two kernels (with different variations) for the initialization part and other two kernels (with different variations) for running the coloring, these kernels are managed and synchronized by the `global::color` function.

2.4.1 Initialization

In the first step of the initialization phase the priorities are calculated, they are used to determine in which order the nodes will be colored. The priorities are calculated in a cuda kernel and are different for each heuristic and they will be explained in the next sections. Afterwards, the `__global__ void calculateInbounds(GraphStruct* graphStruct, unsigned int* inboundCounts, unsigned int* priorities, int n)` is called, it takes as input the priorities previously calculated and produces the `inboundCounts` that are used in the next steps where it helps prioritize the order in which vertices are colored. Moreover, the following data structures are initialized:

1. `d_bitmaps` and `d_bitmapIndex` arrays: they store the available colors of each node, implemented with an array of bool and an index, each vertex can be colored with `d + 1` colors at most, where `d` is the number of neighbors with a higher priority, so the length of the array is `(n + (edgeCount / 2)`.
2. `buffer` and `filledBuffer` arrays: they are used to temporarily store the colors assigned to vertices during the coloring algorithm, allowing for a controlled and synchronized update of the final coloring array.

2.4.2 Coloring

The `while (*uncoloredFlag)` loop is the central part of the graph coloring algorithm. The loop continues until there are no uncolored vertices left in the graph. Inside the loop, there is the **Kernel Execution**: that launches the 'colorWithInboundCountersBitmaps' kernel, which attempts to color uncolored vertices based on inbound counters and updates the buffer and flag accordingly, Synchronizes the device to ensure the kernel completes execution and then Launches the 'applyBufferWithInboundCountersBitmaps' kernel, which applies the buffered colors to the graph, updating inbound counters and bitmaps.

2.5 Random Priority Heuristic

To implement this heuristic the CUDA random number generation library was used, where each thread generate a random priority for a vertex using a seed and a "curand state".

2.6 Largest-Degree-First ordering heuristic

In this ordering heuristic the number of edges incident to each vertex is used to populate the priorities array. This heuristic is widely used because it tends to produce good results in practice.

2.7 Smallest-Degree-Last ordering heuristic

Using this heuristc the priorities are calculated iteratively, the function `SmallestDegreeLast::calculatePriority` use a threshold to assign a lower priority to the vertices with a degree smaller than the threshold and then it adjusts this threshold until all vertices have a priority assigned.

2.8 Saturation Ordering Heuristic

Using this Heuristic, the algorithm iteratively colors the uncolored vertexes whose colored neighbors use the largest number of distinct colors. So the priorities array has to be computer at each iteration of the main algorithm, this process introduces some overheads and makes the algorithm less suitable for a parallel approach.

2.9 Incidence-Degree Ordering Heuristic

Like the previous one, the priorities have to be computed at each step, this time using the number of the colored neighbors.

3 Results and Conclusions

3.1 Devices and Code Comparison

The algorithms were executed using a Tesla T4, based on "Nvidia Turing" architecture, table 1shows the specifications. While the CPU code was executed on an "Intel(R) Xeon(R) CPU @ 2.20GHz".

Table 1: Nvidia T4 GPU Specifications

Specification	**Value**
GPU Architecture	NVIDIA Turing
NVIDIA Turing Tensor Cores	320
NVIDIA CUDA Cores	2,560
Single-Precision (FP32)	8.1 TFLOPS
Mixed-Precision (FP16/FP32)	65 TFLOPS
INT8	130 TOPS
INT4	260 TOPS
GPU Memory	16 GB GDDR6
GPU Memory Bandwidth	300 GB/sec
ECC	Yes
Interconnect Bandwidth	32 GB/sec
System Interface	x16 PCIe Gen3
Form Factor	Low-Profile PCIe
Thermal Solution	Passive
Compute APIs	CUDA, NVIDIA TensorRT, ONNX

3.2 Input Graphs

The following analysis examines three distinct graphs, each characterized by unique values per each features(number of nodes, number of edges, Max degree, etc.). These graphs were obtained from website [1] and chosen because of their popularity at the time this report was written.

The chosen graphs are in the table 2

Graph Name	Nodes	Edges	Max degree	Min degree	Avg degree	Edge Type
youtube-snap	1.1M	3M	28.8K	1	5	Undirected
kron-g500-logn21	2.1M	91M	213.9K	0	86	Undirected
delaunay-n24	16.8M	50.3M	26	3	5	Undirected

Table 2: Choosen Graphs

3.3 Results

In this section, the coloring quality is evaluated and the throughput in colored vertices per second is investigated, wherein the number of vertices is divided by the runtime.

Figure 1 shows the numbers of colors needed by five GPU codes plus one CPU code. Lower is better. The horizontal axis (x-axis) represents the input graphs, while the vertical axis (y-axis) denotes the number of colors, measured on a logarithmic scale. Each version is deterministic, consistently generating the same coloring for a given input.

The Smallest-Degree-Last ordering heuristic uses the smaller number of colors for two of three input graphs, the fact that the "kron-g500-logn21" has a higher average number of degrees could be the cause but it requires further investigations. The one that uses the higher numbers are the two algorithms that use random ordering algorithms: the sequential and the parallel random ordering.

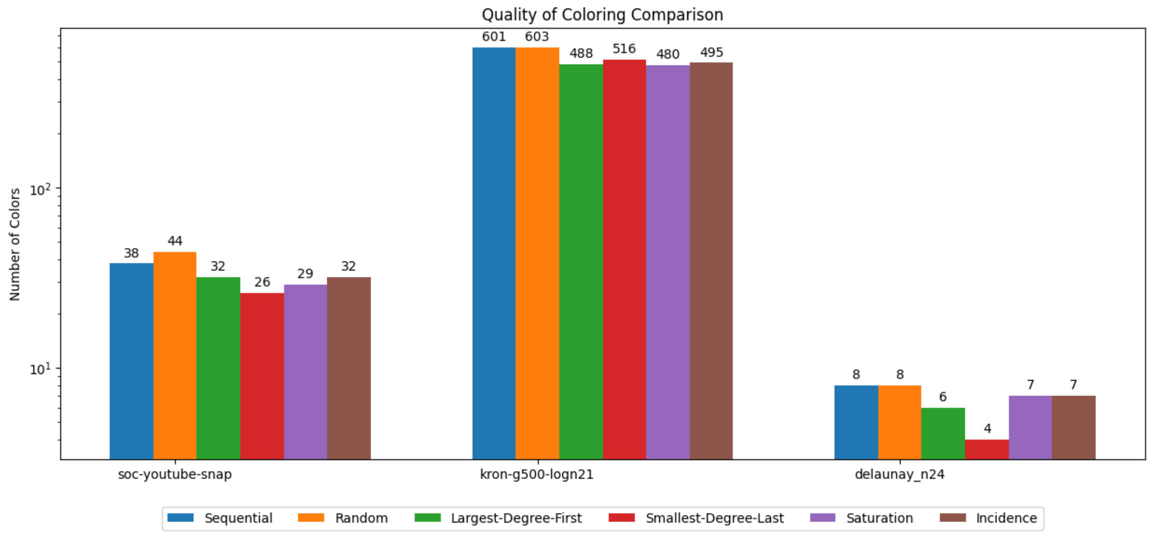

Figure 1: Quality of Coloring Comparison

In the Throughput comparison figure 2 the vertical axis represents throughput measured in millions of completed vertices per second, presented on a logarithmic scale. Throughput, in this context, is a performance metric where higher values indicate better performance. In the graph, only the execution part of the algorithms is considered, while the initialization part can be viewed in the table 3.

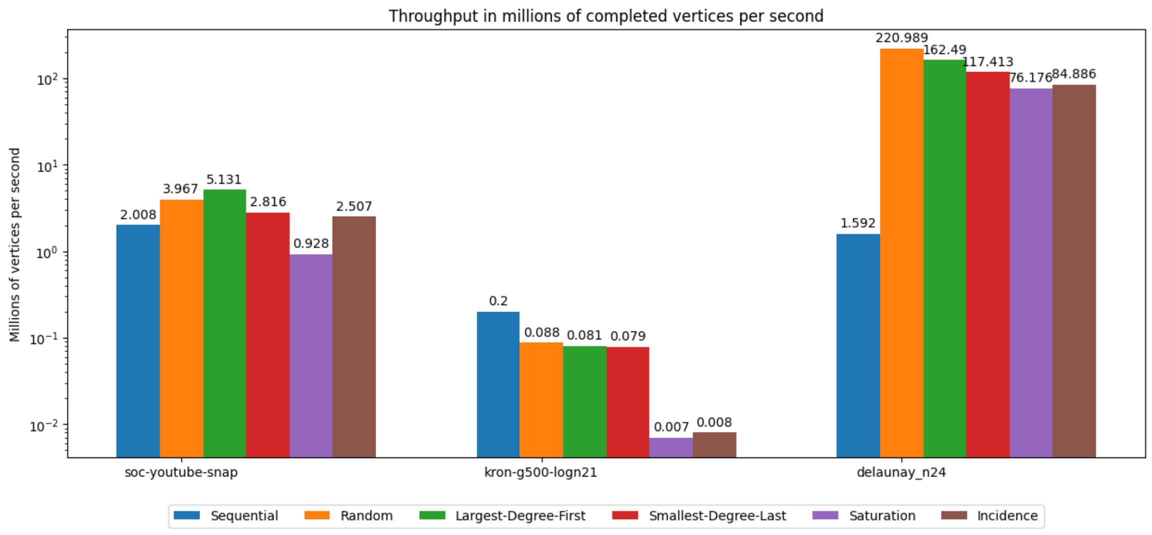

Figure 2: Throughput of Coloring Comparison

Graph Name	Algorithm Name	Init Time (sec.)	Exec Time (sec.)	No. Colors	No. Par. Iterations
soc-youtube-snap	Sequential	0.48481	0.54228	38	0
soc-youtube-snap	Random	0.79535	0.28400	44	261
soc-youtube-snap	LDF	0.72970	0.21000	32	261
soc-youtube-snap	SDL	2.53524	0.39055	26	842
soc-youtube-snap	Saturation	0.63370	1.18416	29	420
soc-youtube-snap	Incidence	0.57260	0.43862	32	283
kron_g500-logn21	Sequential	25.90360	10.49713	601	0
kron_g500-logn21	Random	24.84130	23.78693	603	4307
kron_g500-logn21	LDF	25.10330	25.73980	488	3978
kron_g500-logn21	SDL	173.12700	26.50267	516	4313
kron_g500-logn21	Saturation	25.41770	298.03233	480	3698
kron_g500-logn21	Incidence	25.18650	254.73500	495	3043
delaunay_n24	Sequential	3.24305	10.54690	8	0
delaunay_n24	Random	4.73702	0.07602	8	20
delaunay_n24	LDF	3.78348	0.10339	6	27
delaunay_n24	SDL	3.85813	0.14308	4	44
delaunay_n24	Saturation	3.83603	0.22054	7	29
delaunay_n24	Incidence	3.82566	0.19791	7	22

Table 3: Performance Metrics for Different Algorithms on Various Graphs

The table 3 and the charts 1, 2 shows that parallels algorithms perform particularly better than the CPU with the "delaunay-n24" and "youtube-snap" because their nodes have lower average degrees and a lower variance as shown in 2. The "Smallest-Degree-Last Algorithm" (SDL) with two out of three graphs requires more initialization time and provides better color quality. While the "Saturation" and the "Incidence Heuristics" usually require higher execution time because of the need to update additional data structures in a non parallel way.

3.4 Conclusions

In conclusion, this project has explored and compared parallel graph coloring algorithms implemented in CUDA. The evaluation results highlighted that the effectiveness of the parallelization can vary depend of the graphs characteristics and the used heuristic. These algorithms can be optimized applying differents strategies based of graph feature, for example, processing nodes with similar degree at one time to reduce divergence and analyzing the status of the available color to implement some rules that aim increase parallelism.

References

[1] R. A. Rossi, N. K. Ahmed, The network data repository with interactive graph analytics and visualization (2015).
URL `http://networkrepository.com`