Politecnico di Torino

Master of science program in Computer Engineering

Master Degree Thesis

Improving the scalability of modern applications by parallel multi-core and many-core programming

Supervisor
Quer Stefano

Candidate
Borione Alessandro
id: s272070

December 2022

Abstract

In recent years, the production and usage of vast graphs from different disciplines—social networks, geographical navigation, and internet routing to name a few—has required fast and scalable algorithms. Reachability, single source shortest path, partitioning, and coloring are some of the problems that are commonly applied to graphs. In this thesis, we focus on the problem of graph coloring. To color a graph, we assign a label, called color, to each of its nodes. Colors must be assigned so that no two nodes connected by an edge share the same color. Many scalable algorithms have been proposed; we select, discuss and implement a suite of algorithms for both multi-core CPU and many-core GPU architectures. In particular, we implement the Greedy, Gebremedhin-Manne, and Jones-Plassmann algorithms for multi-core CPU architectures, and the Jones-Plassmann-Luby algorithm for many-core GPU architectures with the help of the CUDA framework. For the latter, we propose a cost-free technique, called index shifting, to lower computation time and reduce the number of colors produced for the solution. We compare the results of our software with cuSPARSE's CSRCOLOR and Gunrock's state-of-the-art implementations, both in terms of computation time and quality of the solution, i.e., the number of colors. We show how our fastest implementation on the GPU produces on average 10% fewer colors than Gunrock's implementation, while also being 2.5 times faster.

Summary

In the last couple of years, governments all around the world have taken action against the COVID-19 pandemic, assisted by mathematical models that produced daily reports to simulate the spread of the virus. To serve a useful purpose, these reports were—and most definitely still are—generated by employing algorithms that, given the input data, can estimate the trend of contagion promptly; moreover, the estimate needed to approximate the real trend, at least for the near future. Modern computer applications often execute complex parallel algorithms on huge amounts of data. It is imperative that such applications can produce an output within a reasonable time frame, which can vary from a few milliseconds to multiple weeks depending on the scale and complexity of the calculation. It is also important that the solution computed by the software is correct, at least to a certain degree, so it can be used to support real-life decisions. When simple parallelization on multi-core architectures is deemed not fast enough, one can resort to many-core Graphical Processing Unit (GPU) programming with the CUDA or OpenCL frameworks. GPUs are massively parallelized processors that can be programmed to exploit the inherent parallelism of a task to massively enhance time performances.

In this thesis, we study the problem of graph coloring, the application of which is central in programs that schedule timetables, allocate registers to variables during compilation, compute derivatives, and many others. Coloring a graph means assigning an integer label, also called a color, to each vertex so that no adjacent pair of vertices is assigned the same color. A visual example that simply shows a graph coloring solution is the way geopolitical maps are colored; by intending regions as vertices so that two confining regions are connected by an edge, the resulting graph can be colored. By then mapping each label to a color, we fill the regions of the original geopolitical map based on how the algorithm assigned the labels. We obtain a geopolitical map with neighboring regions colored differently, as shown in Figure 1. While this is a simple example of graph coloring on a small graph, real applications need to perform colorings on a much larger scale.

Figure 1. Graph coloring on Italian regions

Performing a perfect graph coloring, one where the solution presents the minimum number of colors, is a NP-hard problem and, as such, particularly time-consuming to obtain. In practical applications, it is common to prefer an approximation of the perfect solution, computed by fast and scalable algorithms. We analyze some of the graph coloring algorithms that have been proposed over the years; in particular, we focus on the Greedy, Gebremedhin-Manne, Jones-Plassmann, Jones-Plassmann-Luby, and Cohen-Castonguay algorithms. The greedy algorithm performs coloring with a sequential scan of the nodes; it usually approximates well the perfect solution, but it is slow given its lack of parallelism. The Gebremedhin-Manne approach divides nodes into partitions for parallel execution. The coloring is performed in a greedy-like manner, so conflicts, i.e., adjacent nodes with the same color, may arise; conflicts are corrected sequentially at the end of the execution. As a parallel algorithm, it executes faster than the greedy approach, as long as there are not too many conflicts. The Jones-Plassmann algorithm colors graphs in parallel with a network of processors that communicate via message passing. Each processor generates a random value, and can color its vertex only if its value is the largest among its non-colored neighboring processors. The Jones-Plassmann-Luby and Cohen-Castonguay algorithms color graphs by dividing nodes into independent sets, so that each set can be colored with a single color. For the first algorithm, sets are generated by assigning a random value to each node and iteratively selecting

those nodes whose random values are maximum among their non-colored neighbors. The second algorithm is similar, but uses multiple hash functions to generate multiple random numbers per node, thus finding and coloring more independent sets per iteration.

The analysis is conducted by executing the algorithms, of which we implement different versions and heuristics in the C++ language. In particular, we implement the Jones-Plassmann-Luby algorithm using the CUDA framework, to be executed on GPU devices; moreover, we implement it by applying a heuristic that, to our knowledge, does not appear in literature, to lower computation time and the number of colors generated. We take care, while creating the GPU implementation, to take into consideration performance issues related to high memory throughput—a common characteristic of graph problems on many-core architectures. As the issue cannot be easily, if at all, avoided, we try to keep the memory access operations to a minimum. From the executions, we gather data about execution time and colors used in the solution by running the software on a set of benchmark graphs. Our results confirm that the algorithm running on the GPU is faster than those running on common CPUs. It is interesting, however, how some of the denser benchmark graphs are colored faster by the algorithms implemented to run on the CPU. We also find that the number of colors used to color a graph does not depend on the architecture running the algorithm or its speed, but depends on the algorithm itself. We compare the results achieved by our software with two state-of-the-art implementations from NVIDIA's cuSPARSE library and the Gunrock library, both of which run on GPUs. Our own Jones-Plassmann-Luby implementation proves better than the state-of-the-art implementations, both in terms of execution speed and number of colors used, for the majority of our benchmark graphs.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Modern-day computing requires the execution of algorithms on large amounts of data to solve a specific task, such as environmental simulation or artificial intelligence. Data can be organized in many different ways, depending on what it represents; graphs are one such organization. Graphs are a mathematical construct used to define the relations existing between many objects. In computing, we can use graphs to represent topological information—the layout of roads and streets for vehicular navigation, how atoms are distributed in a molecule, the shape of a network—or more complex data, i.g. the states of a game theory problem, or constraints to consider in parallel execution. As modern problems consider more and more data, graphs grow in size, and faster algorithms are needed to maintain the execution times low. Applying an algorithm to a graph is often a greatly parallelizable task: many algorithms, in fact, repeat the same set of operations on every node. A sequential solution, while simple to code, is expected to be slow. On modern multi-core CPUs, a sequential solution leaves many resources unused, which could be spent to speed the process up. A common and practical solution to implement parallel graph algorithms on a multi-core processor is to partition the vertices in blocks, and assign each block to a child process or thread, that iterates and operates on every node. But as graph instances grow larger, multi-core solutions do not scale well. A more scalable solution is to assign only one node to a process or thread. This solution is still not applicable for very large graphs on common general-purpose multi-core architectures, as the overhead of creating and managing millions of processes or threads would still be too heavy. The solution to this problem comes in the form of Graphical Processing Units (GPU). A GPU is a special processor designed heavily towards concurrency, originally to perform computer graphics operations. GPUs are designed to contain an absurdly large number of cores, which perform the same operations in parallel on different data, implementing the Single Instruction Multiple Data (SIMD) execution model. The idea of offloading computational intensive functions was first explored in the early

2000s. Nowadays, two main frameworks allow GPGPU (General-Purpose computing on GPU): CUDA (Compute Unified Device Architecture) from NVIDIA and OpenCL (Open Computing Language) from the Khronos Group. Details on GPU architecture and GPU programming are reported in Chapter 2.

In conjunction with this thesis, we develop a software suite that implements a number of algorithms to perform graph coloring. We implement the Greedy, Gebremedhin-Manne, and Jones-Plassmann algorithms for multi-core CPU architectures, and the Jones-Plassmann-Luby algorithm for many-core GPU architectures. The software also includes the state-of-the-art implementation of the Cohen-Castonguay algorithm present in the cuSPARSE library. We also analyze the state-of-the-art implementation of the Jones-Plassmann-Luby algorithm from the Gunrock library. Each algorithm is described in Chapter 3, the implementation details are reported in Chapter 4 and the results are discussed in Chapter 5. The software is written in the C++ programming language. C++ is an extension of the C programming language; it originally added support for object-oriented programming, exception handling, generics, and a vast standard library, expanded with further versions of the language. The standard library contains many collection templates for generic programming, as well as many standard algorithms that can be executed on any collection type, regardless of the memory layout. Useful collections classes used in the program are vectors (resizable arrays), maps (collections of key-value pairs), and sets (tree-like strictures that cannot contain duplicate items). Furthermore, C++ provides a platform-independent solution for concurrent programming. We use the thread class to define functions that need to execute concurrently. Another alternative is to use the more recent async-future classes, to define a set of promises that the program executes in background. We choose to use the C++ language as it is a very powerful low-level language that compiles in a fast executable, complete with a vast standard library. We also consider our need of using the CUDA framework, which can be integrated with C++ code, and the simplicity of setting up a working building environment on both Windows and Linux systems with the CMake tool.

Chapter 2

Graphical Processing Unit

Graphical Processing Units are processors heavily geared towards concurrency. The problem a GPU solves lies in the realm of computer graphics, where pixels are printed to a screen to show images. For the image to persist on the screen each pixel, needs to be redrawn at regular intervals, dictated by the screen's refresh rate, which typically is 50 or $60Hz$, but can reach $240Hz$ on high-end devices. At the end of each interval, a new image must be ready for the screen to display, even if the picture is still and has not changed. Modern architectures use a double buffer approach, where one buffer is used as output to the screen, and the second buffer to compute the next frame; when the next frame is ready, the content of the second buffer is copied in bulk to the first. In applications where the image to display on the screen is computed on the fly for every refresh such as rendering of 3D models, the color of each pixel can be computed independently from the other pixels. As they are independent, the computation can be performed in parallel. A normal multi-core CPU is not able, however, to perform a computation for every pixel of a screen in parallel in a single operation: its architecture simply doesn't have enough cores. CPUs, in fact, are optimized to run sequential programs, by allocating a great portion of their chip to cache memories and control circuitry, and using what remains as computing units [1]. On the other hand, GPUs have been designed as chips with the vast majority of their surface dedicated to computing, organized in many small cores. Since neither control logic nor large cache memories contribute to high computational speeds, GPUs can reach GFLOPS (Giga Floating-point Operations Per Second) up to and over three orders of magnitude superior compared to CPUs. Figure 2.1 shows the design differences between CPU and GPU. Another matter taken into consideration by GPU designers is memory throughput. As it happens with CPUs, working with data residing in main memory means first reading the data, performing the computation, and then writing the result. When operating on memory with a GPU, each core reads (or writes) different memory positions simultaneously. GPUs operate on dedicated memory that can reach high throughput speeds to handle a huge number of requests. In general, CPUs are

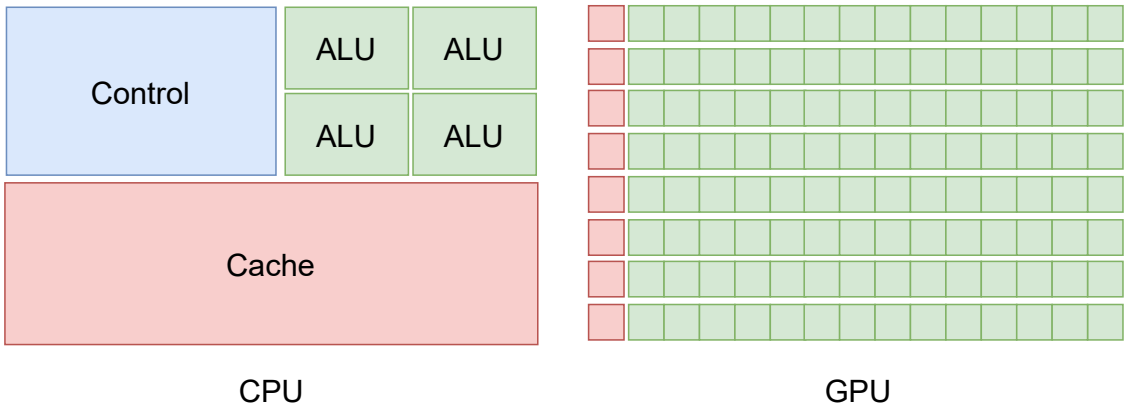


Figure 2.1. CPU and GPU designs.

about 10 times slower than GPUs when considering memory throughput. GPGPU takes advantage of the property of a program known as *data parallelism*. Data parallelism occurs when many operations can be safely performed in parallel on the data structures. Let's take matrix multiplication as an example: given $A \times B = C$, assuming that A is $n \times m$ and B is $m \times n$, C is a $n \times n$ matrix whose elements $c_{i,j}$ are all independent between each other and can be computed concurrently, each by a different core.

2.1 CUDA

CUDA (Compute Unified Device Architecture) is the framework for GPU programming developed by NVIDIA. Before its release in 2007, developers who performed GPGPU could access GPU computing power only through graphics library APIs, which hindered and limited the work that could be achieved. With the release of CUDA, NVIDIA added specific hardware to their chips to handle CUDA requests, so CUDA programs can only run on CUDA-enabled NVIDIA devices.

A CUDA-enabled device contains a set of streaming multiprocessors (SM). In turn, a SM contains streaming processors (SP). SPs are massively threaded and can run thousands of threads in parallel. SPs inside a SM share the same control logic and instruction cache. The dedicated memory is shared between all SMs.

2.1.1 CUDA programming

A CUDA program alternates sections that run either on the host (CPU) or the device (GPU). Usually, sections that present little to no data parallelism are implemented in code that runs on the host, while sections with high data parallelism are

Algorithm 1 How to call a kernel function

```
1: cudaMalloc(...)                                  ▷ Allocate device memory
2: cudaMemcpyHostToDevice(...)                      ▷ Transfer data from host to device
3: kernelFunction<<<nb, nt>>>(...)                  ▷ Launch kernel function
4: cudaMemcpyDeviceToHost(...)                      ▷ Transfer result from device to host
5: cudaFree(...)                                    ▷ Free device memory
```

implemented in code running on the device. The CUDA compiler NVCC accepts source files of unified host and device code. Host code is common C/C++ code that runs on the CPU. Device code is C/C++ code augmented with NVCC-specific keywords to define and launch data parallel functions that run on the device, called *kernel functions*, or simply *kernels*. When a kernel function is called, it spawns a large number of threads to be run on the GPU. Threads are organized in blocks, and threads within the same block can cooperate. Blocks organize threads as a 3-dimensional array, assigning a 3-dimensional index to each thread for recognition. Blocks are further contained in a 3-dimensional grid. A kernel launch spawns a single grid of blocks. The dimensions of the grid and blocks are passed during the kernel launch with the special syntax $<<< nb, nt >>>$ to be infixed between the kernel name and the list of arguments, where nb is the number of blocks in a grid, and nt is the number of threads in each block. nb and nt are of type DIM3, which is a struct of three unsigned integer values x, y, z. nb and nt can also be 1- or 2-dimensional, by appropriately setting y and z to 1. A typical host code to launch a kernel is reported in Algorithm 1. Because the GPU can only operate on its dedicated global memory, data must be moved between host and device before launching a kernel function, and from device to host after it has finished, to retrieve the result. Memory on the device must be allocated before and freed after use, similarly to how it is done normally with heap memory, with cudaMalloc and cudaFree functions. Transfer of data is achieved by calling the functions cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or the more general cudaMemcpy. The list of parameters accepted by these functions can be checked in the CUDA user guide [2].

CUDA provides three qualifier keywords to mark functions. The *__global__* qualifier marks a kernel function; a kernel function can only be called by the host code, but runs on the CPU by spawning a grid of threads. The *__device__* qualifier marks a function that runs on the device and can only be called by a kernel function of another device function. The *__host__* qualifier marks a function that runs on the host and can be called only by another host function; in other words, a host function is a traditional C/C++ function. The *__device__* and *__host__* qualifiers can be combined on the same function declaration. The compiler generates two versions of said function: a host function and a device function.

A grid of blocks contains blocks of threads organized in a 3-dimensional array. Also, blocks of threads contain threads organized in 3-dimensional arrays.

Both blocks and threads are indexed as if they are placed in a 3-dimensional space with 3-dimensional coordinates. CUDA pre-initializes the constants *blockIdx* and *threadIdx* with the correct coordinates of a running GPU thread. The dimensions of the grid and blocks specified at the launch of the kernel are also available via the constants *gridDim* and *blockDim* respectively. These four constants are only available inside a kernel function and contain the attributes x, y and z. Coordinates start at (0,0,0) and cover the space up to $(dim.x - 1, dim.y - 1, dim.z - 1)$ for both threads and blocks. Thread coordinates are unique inside a block, but each block contains threads with the same coordinates as the other blocks. It is often useful to index a thread with a scalar index rather than vectorial coordinates. To do so, we compute the cross product between the row vector $threadIdx$ and the column vector constructed as $(1, blockDim.x, blockDim.x \times blockDim.y)^\mathsf{T}$: the result is the scalar $tIdx_B$, unique for each thread inside a block. The same can be achieved for a scalar block index $bIdx$, by cross-multiplying the row vector $blockIdx$ with the column vector $(1, gridDim.x, gridDim.x \times gridDim.y)^\mathsf{T}$. It is also possible to find a global index for a thread, unique in all the grid, with the formula $tIdx = tIdx_B + bIdx\,(blockDim.x \times blockDim.y \times blockDim.z)$. Both 3-dimensional coordinates and scalar indexes can be used to divide the work each thread is tasked to do, usually by accessing different portions of input and output arrays.

2.1.2 Memories

CUDA programs rely on different types of memories with varying characteristics to achieve speedup on a program. The types of memory are *registers*, *shared*, *constant*, and *global* memory.

Registers are fast memory, automatically allocated to contain the scalar variables declared in the kernel function. NVCC introduces qualifier keywords to denote different types of memory. Register memory is private to the thread, and is allocated for each thread in the grid.

Shared memory is fast and, as the name suggests, is shared among multiple threads. Threads within the same block can access the same shared memory portion with read and write operations. Shared memory is declared inside the device code with the keyword ___*shared*___. Register and shared memory are freed at the end of the kernel execution.

Constant memory is a memory shared by all threads of all grids. It can reach particularly high speeds when all threads access the same cell. Constant memory is read-only from the device point of view, but can be written by the host code, which is also tasked with the declaration with the keyword ___*constant*___. Constant memory is very limited in modern GPUs, all mounting 65536 bytes.

Global memory is the most common and most abundant type of memory in a GPU, usually spanning between hundreds of Megabytes and tens of Gigabytes.

It can be accessed by all threads of all grids, but is slower than the other types of memories. It is also used to store the local array variables declared inside a kernel function. Global memory is declared by the host code, Constant and global memories are maintained between multiple kernel calls and are only de-allocated either at the end of the program or when the host code frees them.

2.1.3 Kernel scheduling

As explained earlier in the chapter, launching a kernel spawns many threads to fill the grid, given the dimensions nb and nt. The total number of threads is given by the multiplication $nb.x \times nb.y \times nb.z \times bt.x \times nt.y \times nt.z$. Generally, the number of threads can get quite high, and GPUs do not have the resources to run all of them at the same time, so a scheduling approach must be implemented. We have seen how a GPU is comprised of Streaming Multiprocessors (SM), each of which contains several Streaming Processors (SP). Each SP is tasked with the execution of a block of threads; when a SP terminates the execution of a block, another one is scheduled on that SP, as long as there are blocks that need to execute. Hardware limitations put a maximum to the number of threads that can be scheduled in a single SM at the same time. Let n_{SP}^{SM} denote the number of SPs for every SM, and n_t^{SM} the number of threads per SM—both of which are fixed values decided by the manufacturer of the GPU—the number of threads that should be contained in each block to achieve maximum parallelization is n_t^{SM}/n_{SP}^{SM}. Other attributes determine how many blocks can be scheduled concurrently, such as register occupation per thread and shared memory occupation per block.

As it has been described so far, scheduling on the GPU has a major flaw: when an instruction needing many clock cycles to complete—such as a floating point division or a load from global memory—is issued, the computational resources of the SPs are idle, waiting for the result. Instead, CUDA devices are designed to divide each block in *warps* of 32 consecutive threads. Warps are scheduled and run in the SPs, each thread of a warp executing the same instructions at the same time, in a Single-Instruction Multiple-Thread (SIMT) execution style. When a warp issues a long-running instruction, the idling warp is scheduled out of the SP, and a ready warp takes its place. In this way, since swapping of warps is a low-cost operation in a GPU, the execution hardware can be kept busy at all times while the long-latency operations execute in background. The technique of swapping idling warps with ready warps to keep the resources fully utilized is called *latency hiding*. The ability to hide the effect of long-latency memory load instructions is the main reason why GPUs do not need to dedicate as much chip area to cache memories and branch prediction hardware as CPUs do.

When a warp executes a conditional branch instruction, three main outcomes may occur: all threads take the branch and continue execution on a new instruction, no thread takes the branch and the execution continues on the following instruction,

or some threads take the branch and other threads don't take the branch. In the latter case, we say the threads diverge. War execution sequentially follows all diverging paths, as long as at least one thread follows it. To avoid computational errors due to executing instructions in the wrong branch, threads that do not follow the branch currently being executed get deactivated. Divergence happens within control flow instructions such as *if-then-else* constructs and loops, depending on the state of the execution and the conditional being evaluated. For example, if we consider a warp executing threads with indexes 0-31, executing the instruction

$$\textbf{if}\ (threadIdx.x < 16) \quad \ldots \quad \textbf{else} \quad \ldots$$

divides the warp in half. The warp executes the *then* block and the *else* block sequentially. While executing the *then* block, threads 16-31 are deactivated. Similarly, threads 0-15 are deactivated while executing the *else* block. This is cause for divergence, as only part of the computational resources is utilized, given that some threads are not active. On the other hand, the same *if* conditional does not cause divergence in the warp with threads 32-63, as all threads follow the *else* block. Execution time suffers from divergence, as all diverging paths are executed sequentially.

2.2 OpenCL

OpenCL (Open Computing Language) is a standardized set of cross-platform APIs for parallel computing, first proposed by Apple and developed by the Khronos Group. Many concepts of OpenCL programming are similar to CUDA, while other more complex aspects are present to manage different devices due to the cross-platform scope of the standard. OpenCL contains many optional features to allow compatibility with a broader set of devices. Because of this, cross-platform software usually contain many code paths that produce the same result employing different means, so that a device can choose at runtime which one to run based on its characteristics. The data parallelism model of OpenCL differs from the CUDA one by the names of its components. OpenCL programs are characterized by a mixture of device code and host code. The host code manages the execution of the device code by launching kernel functions. The launch of a kernel function causes the spawn of *work items*, the OpenCL correspondence of CUDA threads. Work items are organized in *work groups*, the same as CUDA blocks, and are identified by global dimension index ranges called *NDRanges*, which take the role of CUDA grids. The way work items are indexed in OpenCL is slightly different from the indexing of threads in CUDA. A work item running a kernel function can access its global index by calling the function *get_global_id()*, and passing the values 0, 1, or 2 to specify the dimension of the index returned. For example, *get_global_id(0)* corresponds to the CUDA code $blockIdx.x \times blockDim.x + threadIdx.x$. Passing

1 or 2 as values returns the index from the y or z dimension instead. The local index of a work item inside a work group can be accessed by calling *get_local_id()*, again with parameters 0, 1 or 2 to specify the x, y or z dimension; this directly corresponds to *threadIdx* in CUDA. To get the dimension in the x dimension of the NDRange, we call *get_global_size(0)*. This API returns the number of work items, so the corresponding CUDA code is $gridDim.x \times blockDim.x$. To know the dimension of the work groups, we use *get_local_size()* with 0, 1, or 2 to specify the dimension; it corresponds to *blockDim*.

OpenCL devices run the kernel functions of an OpenCL program. Each device is comprised of *Compute Units* (CU), which correspond to CUDA streaming multiprocessors, but can also represent the cores of a CPU or other executions units in FPGAs. Each CU is further comprised of *Processing Elements* (PE), which correspond to CUDA streaming processors.

OpenCL exposes a hierarchy of memories similar to CUDA. OpenCL *global memory* corresponds to the CUDA global memory. It can be dynamically allocated by the host program and allows read/write access by both host and device code. *Constant memory* in OpenCL supports read/write operations from the host but is read-only from the device. Unlike CUDA constant memory, the size is not fixed to 65536 bytes, as many more devices are supported; the actual size of OpenCL constant memory is available via an API call. OpenCL *local memory* corresponds to CUDA shared memory. Local memory can be dynamically allocated by the host and statically allocated by the device. Finally, OpenCL *private memory* corresponds to CUDA *local memory*. The last two types of memory cannot be accessed with read/write operations from the host code.

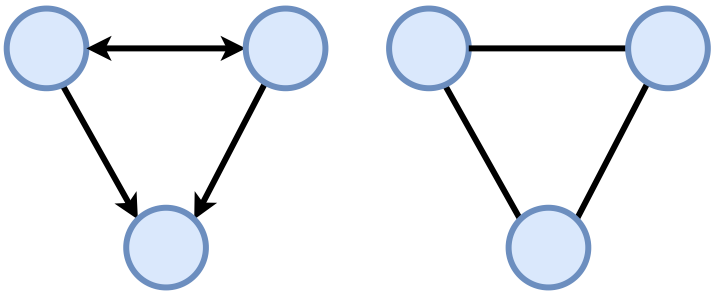
Kernel functions in OpenCL have the same structure as in CUDA, but use different keywords. To indicate a kernel function, a function declaration is marked with the *___kernel* qualifier, which serves the same purpose as *___global___* does in CUDA. Pointers to global device memory accepted by a kernel function must be marked with the keyword *___global* in the function definition.

Device management is more complex in OpenCL than in CUDA, as the former supports more devices in terms of architectures, vendors, and functionality than the latter. A typical host code gets the ids of the devices present on the system with the *clGetDeviceIDs()* API function, and uses the returned information to create a context calling *clCreateContext()*. Contexts are used in OpenCL to manage a device. The host will then create a command queue for a device with the function *clCreateCommandQueue()*; command queues are used to instruct devices on the computation to perform. The host can then submit work for a device through its command queue, like allocating and initializing an array in global memory, launching a kernel, and copying an array from device to host memory. The specific API calls and the parameters accepted are outside the scope of this thesis and can be referenced in the OpenCL programming guide [3].

Chapter 3

Graphs: Notation and Coloring

Graph theory is the branch of mathematics that studies pairwise relations between objects. The relations are modeled as a graph $G = (V, E)$, where V is the set objects, also known as *vertices* or *nodes*, and $E \subseteq \{(v, w) \mid (v, w) \in V^2\}$ is the set of *edges*, modeled as ordered pairs of nodes. With this definition, the relation described by the graph is valid from v to w, but not vice versa. A graph defined as such is called a **directed graph**. If the relation described by the graph is intrinsically bidirectional, the definition of E can change to $E \subseteq \{\{v, w\} \mid v, w \in V\}$, where every item is an unordered pair of nodes; this type of graph is called an **undirected graph**. An edge $e_L = \{v, v\}$ expresses a relation the node v has with itself, and takes the name of **loop** or **looping edge**. For a directed graph, the number of edges entering and exiting a node v is called **in-degree** and **out-degree** respectively. For an undirected graph, in-degree and out-degree are equivalent, and simply called **degree**.



(a) A directed graph (b) An undirected graph

Figure 3.1. An example of a directed (a) and an undirected (b) graph.

A **Directed Acyclic Graph** (**DAG**) is a directed graph with no cycles. Because it has no cycles, it is not possible to follow a path that visits the same node twice. In a DAG, it is possible to follow the edges starting from one or more nodes with in-degree equal to 0 called *sources*, arriving at one or more nodes with out-degree equal to 0 called *sinks*.

Graphs carry an intrinsic property called **density**, defined as the number of edges of the graph divided by the total number of possible edges. For directed graphs, the density is $D = \frac{|E|}{|V|(|V|-1)}$, and for undirected graphs it is $D = \frac{2|E|}{|V|(|V|-1)}$. We say a graph is **dense** when its density is close to its maximum value $D_{max} = 1$, and **sparse** when its density is close to its minimum value $D_{min} = 0$. The distinction between sparse and dense graphs is not strictly defined and may change depending on the context.

A set of nodes $I \subseteq V$ is **independent** if the nodes in it do not share any edge. A **maximal independent set** (**MIS**) is an independent set that is not a proper subset of a larger independent set.

3.1 Notation

The following notation will be used in the next chapters. We will use n and m as the cardinalities $|V|$ and $|E|$ respectively. The minimum, maximum, and average degrees across a single graph are represented as δ, Δ, and $\overline{\delta}$. The degree of a vertex v for an undirected graph is $\delta(v)$. The set of nodes that share an edge with a node v—adjacent or neighbors to v—is denoted by $N(v)$ and is defined as $N(v) = \{w \mid (v, w) \in E\}$ for a directed graph and as $N(v) = \{w \mid \{v, w\} \in E\}$ for an undirected graph.

3.2 Representation in computer memory

There are many ways to store a graph inside the main memory of a computer.

The **coordinates** (**COO**) format uses a list of pairs of integers to store every element $(v, w) \in E$. The two integers represent v and w, and the list is m pairs in length. The COO format occupies $2m$ memory cells and may be inconvenient with certain operations, especially if the elements are not properly sorted. Assuming no use of indexing data structures to support the navigation, iterating through the adjacencies of a node requires a binary or linear search of the array, depending on if the elements are sorted or not respectively.

The **adjacency matrix** format uses a $n \times n$ matrix—an array of arrays—to store information about the edges of a graph. Each row and each column represent a vertex of the graph, and each cell of the matrix represents whether an edge connects the row vertex to the column vertex. Despite being easy to implement and manage,

n^2 cells are used to store m elements; therefore a matrix should only be used for dense graphs, where m is not much less than n^2.

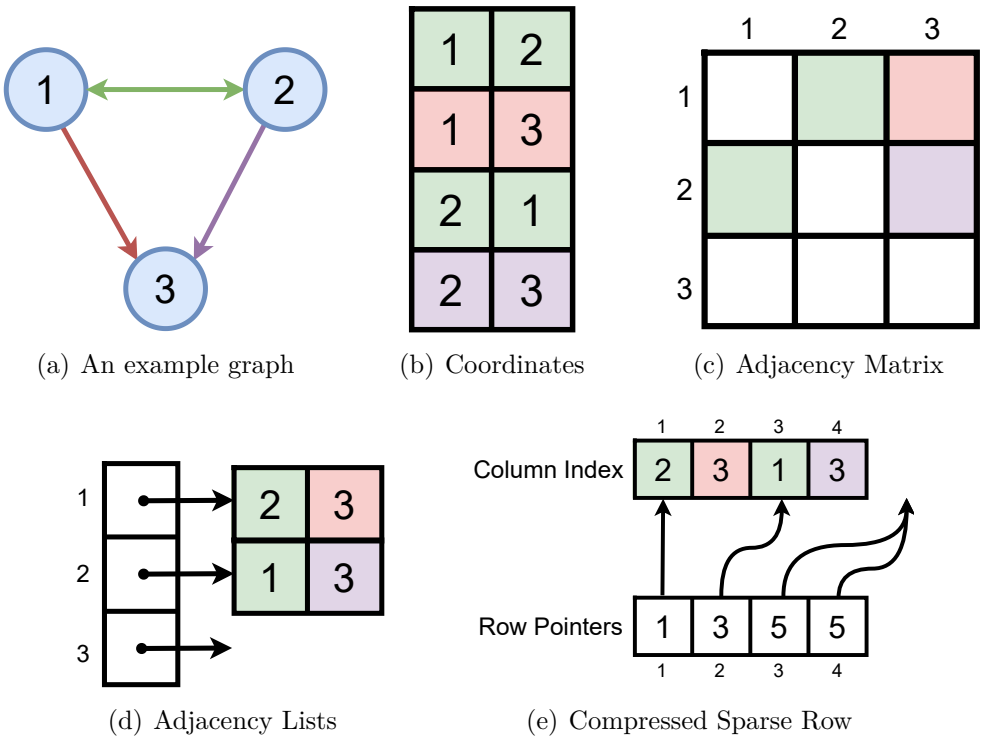
Similar to the adjacency matrix, **adjacency lists** store information about the graph's edges. For every vertex, an array lists all its neighboring vertices, and all the arrays are indexed in an encompassing array. This method is preferable for sparse graphs—where $m \ll n^2$—as the encompassing array indexes n lists that, added up, occupy m memory cells, for a total of $n + m$ memory occupancy.

Adjacency lists are not necessarily stored sequentially in memory and may cause the cache miss rate to increase when accessing the data structure. The **compressed sparse row** (**CSR**) format reduces this problem by adopting storage with only two arrays. The first, called *column index*, is the concatenation of all adjacency lists in a single array. The second, called *row pointers*, stores for each vertex a pointer to the column index array where its adjacency list begins. To iterate through $N(v)$ one first accesses the row pointers array at index v to load index p_v, and index $v + 1$ to load index p_{v+1}. The sub-array of the column index array from index p_v to index p_{v+1} (with the end excluded) is the adjacency list corresponding to vertex v. To implement this process without edge cases, it is common to add one extra cell at the end of the row pointers array, with value $m + 1$ in 1-indexed systems (or value m in 0-indexed systems). This extra cell contains a pointer that points outside the bounds of the column index array; this is however not a threat, since the iterating algorithm should never access that particular cell, even without extra checks. Figure 3.2 shows the formats described above for a simple example graph. In the example, colors represent the correspondence between memory cells and edges.

When working with undirected graphs, it is convenient to convert each unordered pair $e = \{v, w\} \in E$ in two ordered pairs $e' = (v, w)$ and $\overline{e'} = (w, v)$, and assume the graph is directed. A looping edge $e_L = \{v, v\} \in E$ only produces one ordered pair $e'_L = (v, v)$ for the purpose of this transformation. In the next chapters, all undirected graphs will be assumed to have already been transformed, and the value of m will represent the cardinality of the set of edges for the resulting directed graph.

3.3 Graph Coloring

Graph coloring is a problem applied to undirected graphs and consists in finding a function $C : V \to \mathbb{N}$, while keeping the constraint $C(v) \neq C(w), \forall (v, w) \in E$ true. In other words, the solution requires mapping each vertex to an integer value—also called a color—so that there is no edge between two vertices with the same color. For the purpose of graph coloring, we do not consider looping edges, as they connect a node with itself, and this would render the node impossible to color. Many problems can be solved using graph coloring, such as timetable scheduling [4], register allocation in compiler optimization [5], Sudoku solving [6], and the approximation



(a) An example graph (b) Coordinates (c) Adjacency Matrix

(d) Adjacency Lists (e) Compressed Sparse Row

Figure 3.2. Example of data structures to store graphs. Coordinates (b), adjacency matrix (c), adjacency lists (d) and compressed sparse row (e) representations for graph (a).

of Jacobians and Hessians during automatic differentiation [7, 8]. Graph coloring can be also useful when assigning resources during parallel execution; considering tasks as nodes, and constraints of execution as edges, tasks with the same color can always be executed in parallel. To help solve these and other problems it is important to develop an algorithm that can find the optimal solution $C_O(G)$, which presents the minimum number of colors $\chi(G)$. $\chi(G)$ is known as the *chromatic number* and is the minimum number of colors needed to fully color graph G. Unfortunately, finding the optimal solution to a graph coloring problem is known to be of class NP-hard [9]. As such, an exact, optimal solution cannot be easily found. Instead, research has been focusing on developing fast heuristics that closely approximate $C_O(G)$, are fast to compute and, possibly, parallelizable.

3.4 Related work

The easy way to perform graph coloring is by using a *greedy* approach. One simply selects a vertex and colors it with the best possible color available, repeating this

process until all vertices are colored. In general, the best color for a node is the lowest color not used by one of its neighbors. Choosing a color with this method means that a node v will always be assigned a color lower or equal to $\delta(v) + 1$, because only its δ neighbors need to be considered. The order in which the nodes are colored is key to the quality of this approach. While an ordering that produces $\chi(G)$ colors exists, there is not a programmatically good way of finding such ordering. An arbitrary ordering produces at most $\Delta+1$ colors: this is the case where the node with the maximum degree is colored after all its neighbors, all of which are colored with Δ different colors. Different heuristics have been proposed to iteratively choose the next vertex to color so that the number of colors is kept low. The **Largest Degree First** (**LDF**) [10] ordering colors the nodes based on their degree: the vertices are ordered by descending degree and colored in sequence. This ordering tries to reduce the possibility of using many colors, by first coloring the nodes that have the possibility of being assigned high colors, that is nodes with a high degree. The **Incidence Degree** (**ID**) [7] ordering colors the nodes by descending incidence degree. The incidence degree of a vertex is defined as the number of its colored neighbors. The **Saturation Degree** (**SD**) [11] ordering colors the nodes by descending saturation degree. The saturation degree of a vertex is defined as the number of different colors assigned to its neighbors. These three heuristics color the "problematic" vertices first—to avoid possible high colors later on—but define being "problematic" in different ways. In general, SD produces the best results, followed by ID, and then by LDF. Given the sequentiality of the greedy approach, it is difficult to parallelize it.

Gebremedhin and Manne [12] develop a parallel algorithm to perform graph coloring by partitioning V in blocks to be colored separately. Since V is partitioned arbitrarily, there is no guarantee that the graph is colored correctly, as two neighboring nodes v and w may be assigned to different blocks and given the same color. The algorithm overcomes this problem by allocating some time at the end of the execution to search for errors in the coloring, and re-coloring the offending nodes. They also propose an improvement for this algorithm in the number of colors used, at the cost of a second coloring step.

A different approach follows the observation that a set of independent nodes can be colored in parallel. The observation stems from the research by Luby [13], in which he proposes a parallel algorithm to find maximal independent sets of nodes. The MIS is found by first selecting an independent set, and iteratively augmenting it to become maximal. The initial independent set is created by randomly choosing vertices with a probability proportional to the inverse of their degree. Jones and Plassmann [14] use non-maximal independent sets to color nodes with the lowest color not used by one of its neighbors. The independent sets are created by choosing only nodes whose value is the local maxima between its neighbors; the value of each node is randomly generated in advance. Combining these two

different formulations, the Jones-Plassmann-Luby (JPL) algorithm finds an independent set per iteration, coloring its nodes with the same color in parallel, until all nodes are colored. An extension to this algorithm by Cohen [15] proves that the best approach is to find two different independent sets—of local maxima and local minima—per iteration, which are disjoint per definition and can be colored in parallel. Jones and Plassmann [14] also propose two parallel algorithms to perform graph coloring by message passing between multiple processors. Cohen and Castonguay [16] propose and implement the state-of-the-art routine CSRCOLOR included in the cuSPARSE library for CUDA-enabled devices; the algorithm follows a modification of the JPL algorithm that uses multiple hash functions to generate and color multiple independent sets per iteration. Naumov *et al.* [17] compare the CSRCOLOR implementation with a CUDA implementation of the JPL algorithm, reporting that the former is roughly 3 to 4 time faster, but generates 2 to 3 times more color than the latter. Osama *et al.* [18] implement the JPL algorithm in the GraphBLAST and Gunrock frameworks for GPU programming and compare the results. They find that the fastest version of the algorithm runs on Gunrock without any form of load balancing, finding two independent sets per iteration.

In the following sections, we discuss the formulations of these few parallel algorithms and provide some insights into the state-of-the-art implementations.

3.5 Gebremedhin-Manne algorithm

The Gebremedhin-Manne [12] algorithm is a block algorithm to perform graph coloring in parallel. The pseudo-code for the algorithm is reported in Algorithm 2, and can be decomposed in three steps: pseudo-coloring (lines 1-4), conflict search (lines 5-13), and conflict resolution (lines 14-16). In the pseudo-coloring step, each vertex is assigned a color. First, V is partitioned in p blocks $V_1, V_2, \dots V_p$, each containing n/p nodes, assuming $n/p \in \mathbb{N}$ for the sake of simplicity. Each block is then assigned to a processor, which has the task of coloring it. The pseudo-coloring is performed synchronously, meaning that, if processor P_i has already colored l nodes from its assigned block V_i, it can start to color its $l + 1$th node only if all other processors have also already colored l of their nodes. This synchronicity divides the time for the pseudo-coloring step in n/p frames. At each frame, p nodes are colored, one from each block. The coloring is performed as usual, by choosing the smallest color not used by the neighbors of the node to color. The time frame division also helps in reducing the number of neighboring nodes in choosing a color, as only the ones colored in previous frames are to be considered; all the others are not yet colored, so they can be skipped without affecting the result. In the case in which two adjacent nodes are colored during the same time frame, they may be assigned the same color. However, it is not a problem because the next steps will take care of any conflicting colors. It is also the reason why the first step is called pseudo-coloring: the "pseudo" prefix signifies that the coloring is not perfect, and

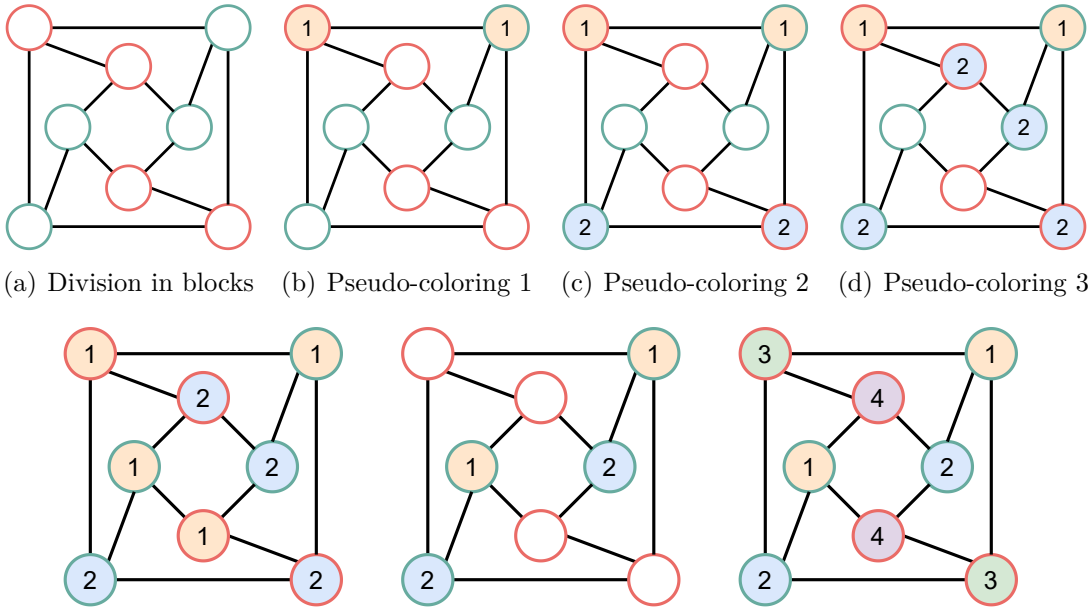
Algorithm 2 Standard Gebremedhin-Manne Algorithm

GEBREMEDHIN-MANNE-STANDARD $(G = (V, E), colors)$

1: **for** $v \in V$ in parallel **do**
2: $\quad colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(w) \mid w \in N(v)\}$
3: $\quad$ Barrier wait
4: **end for**
5: $K \leftarrow \emptyset$ ▷ Set of nodes that need to be recolored
6: **for** $v \in V$ in parallel **do**
7: $\quad S \leftarrow$ nodes colored in the same step as v on line 2
8: $\quad$ **for** $w \in (N(v) \cap S)$ **do**
9: $\quad\quad$ **if** $colors(v) = colors(w)$ **then**
10: $\quad\quad\quad K \leftarrow K \cup \min\{v, w\}$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: **end for**
14: **for** $v \in K$ **do**
15: $\quad colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(w) \mid w \in N(v)\}$
16: **end for**

may contain errors.
After all the nodes are colored, it is the conflict search step. This step aims at identifying all conflicts in the pseudo-coloring generated in the previous step. Conflict search is still performed in parallel, with the same p blocks as the first step. Each processor will check all nodes in its block only against their neighboring nodes which were colored during the same time frame in the previous step. If a conflict is detected, the node with the lower index is uncolored and saved in a global array for the next step.
The last step is the conflict resolution step. All conflicting nodes found and uncolored during the second step are correctly assigned a new color. This step is performed sequentially, to avoid causing further conflicts. Figure 3.3 shows an example of a graph colored following the algorithm. The graph is initially partitioned into two blocks, denoted by the different border colors (a). Each time frame, a node per block is pseudo-colored (b), (c), (d), (e). During the conflict search, equal adjacent colors are removed from the graph (f). Nodes that are now uncolored are assigned a new, non-conflicting color sequentially (g).

The algorithm just described takes the name of standard Gebremedhin-Manne algorithm. The original authors also propose an improved version, with the intent of finding a solution with fewer—or at most the same—colors. The improved Gebremedhin-Manne algorithm presents a second pseudo-coloring step, immediately before the conflict search step. Algorithm 3 shows the extra code to implement the second pseudo-coloring step for the improved algorithm, to be inserted between lines 4 and 5 of Algorithm 2, assuming that the first step is performed on the array $colors'$ instead of $colors$. The colors assigned in the first pseudo-coloring



(a) Division in blocks (b) Pseudo-coloring 1 (c) Pseudo-coloring 2 (d) Pseudo-coloring 3

(e) Pseudo-coloring 4 (f) Conflict search (g) Conflict resolution

Figure 3.3. Gebremedhin-Manne algorithm applied to a simple graph.

Algorithm 3 Second step of improved Gebremedhin-Manne algorithm.

IMPROVED-PSEUDO-COLOR $(G = (V, E), colors, colors')$

1: **for** k varying from $\max_{v \in V} colors'(v)$ down to 1 **do**
2: $\quad ColorClass \leftarrow \{v \in V \mid colors'(v) = k\}$
3: $\quad$ **for** $v \in ColorClass$ in parallel **do**
4: $\quad\quad colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(w) \mid w \in N(v)\}$
5: $\quad$ **end for**
6: $\quad$ Barrier wait
7: **end for**

step are used to define color classes, on line 2. A color class is a set of vertices that share the same color. Notice that at this point, a color class is not an independent set, because conflicts are not yet corrected. The pseudo-coloring is then repeated for each color class. By coloring the color class that corresponds to the highest color first, and going down to the lowest color, Gebremedhin and Manne show that the number of expected conflicts decreases. Moreover, the number of colors also decreases—or remains the same—with respect to the first pseudo-coloring step.

The original paper also describes how to modify the algorithm to be asynchronous. This is an important observation, as it is shown that the synchronous algorithm is very slow due to the different loads of each processor during the first pseudo-coloring step. To make the algorithm asynchronous, simply remove the

barrier synchronization on line 3 of Algorithm 2. Without synchronization, however, the pseudo-coloring step does not present distinct time frames anymore. This means that in the asynchronous algorithm it is not possible to reduce the number of adjacent nodes checked during the pseudo-coloring and the conflict search steps. In general, the asynchronous algorithm is faster than the synchronous algorithm, but it produces slightly more colors.

3.6 Jones-Plassmann-Luby algorithm

An independent set is a set of items independent between each other. Nodes in a graph are independent if are not connected by an edge. So, as noted by Luby [13], an independent set of nodes can be colored in parallel using the same color. An outline of the process is shown in Algorithm 4. Choosing an independent set can be done

Algorithm 4 Overview of coloring via independent sets

```
INDEPENDENT-SET-COLOR (G = (V, E))
1: V' ← V
2: while V' ≠ ∅ do
3:     Choose an independent set I from G' = (V', E)
4:     Color I in parallel
5:     V' ← V' \ I
6: end while
```

in different ways. The sequential approach is at least $O(m)$, so is not ideal on large graphs. Examining Algorithm 4, it is evident that if it is possible to find the largest independent set possible at every iteration, the number of iterations decreases, thus speeding up the coloring. The largest independent set takes the name of Maximal Independent Set (MIS), defined as an independent set that is not a proper subset of a larger independent set. A simple greedy approach to choose a MIS is to scan each $v \in V$ and add it to I, only if I does not contain any adjacent nodes to v. This approach produces a particular MIS known as Lexicographically First MIS; unfortunately, the algorithm is inherently sequential, and cannot be parallelized. Luby [13] presents an algorithm to solve the MIS problem. Luby's algorithm works by first choosing an independent set and augmenting it to a maximal independent set, as shown in Algorithm 5. Choosing an independent set from the graph on line 5 can be performed in different ways. Luby proposes both a non-deterministic and a deterministic approach that can be executed in parallel, and are out of the scope of this thesis. Jones and Plassmann use a Monte Carlo rule that produces an independent set from a graph using an array of random values ρ of length n. Vertex v can be added to the independent set I if the random value $\rho(v)$ is the local maxima between the random values $\{\rho(w) \mid w \in N(v)\}$ assigned to the neighbors of v; vertices already colored are ignored during this computation. Algorithm 6

Algorithm 5 Luby's algorithm for Maximal Independent Set.

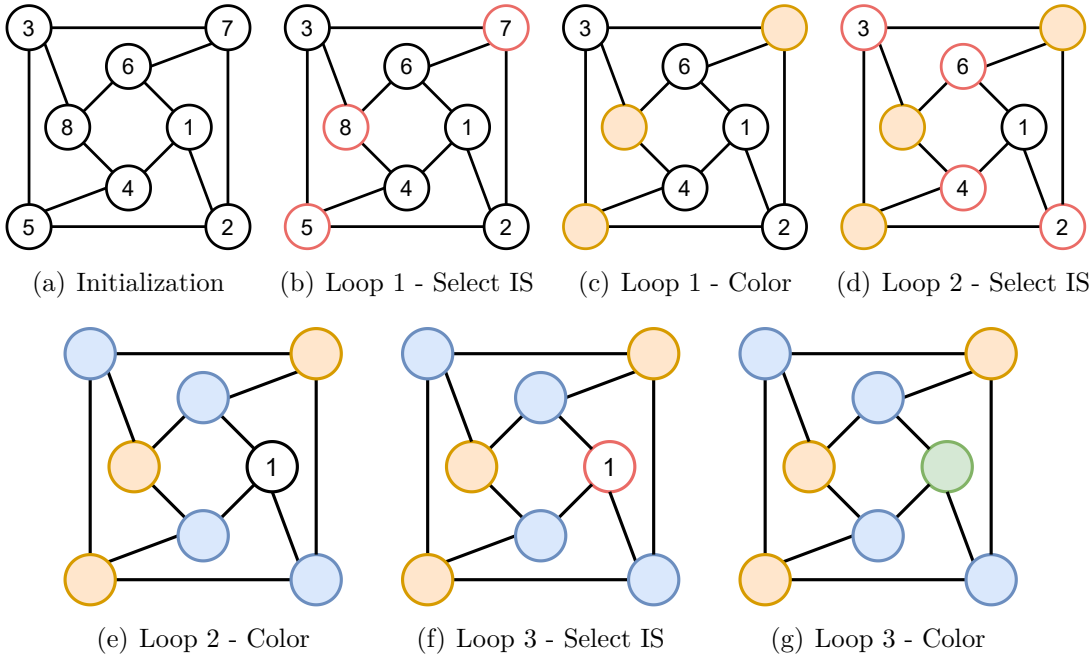
MIS-Luby $(G = (V, E))$
1: $I \leftarrow \emptyset$
2: $V' \leftarrow V$
3: $G' \leftarrow G$
4: **while** $G' \neq \emptyset$ **do**
5: Choose an independent set $I' \in G'$
6: $I \leftarrow I \cup I'$
7: $X \leftarrow I' \cup N(I')$
8: $V' \leftarrow V' \setminus X$
9: $G' \leftarrow G(V')$ ▷ subgraph of G induced by V'.
10: **end while**
11: I is a maximal independent set

shows the pseudo-code for this rule. The color Jones and Plassmann assign to I is chosen separately for each vertex as the smallest available color not already used by one of its neighbors. By finding a middle ground between Luby's algorithm and the

Algorithm 6 Rule to choose an independent set

Jones-Plassmann-Find-IS $(G = (V, E), \rho)$
1: $I \leftarrow V$
2: **for** $v \in V$ in parallel **do**
3: $X \leftarrow \{w \mid \rho(w) > \rho(v), w \in N(v)\}$
4: **if** $X \neq \emptyset$ **then**
5: $I \leftarrow I \setminus \{v\}$
6: **end if**
7: **end for**
8: I is an independent set

one proposed by Jones and Plassmann, we define the Jones-Plassmann-Luby (JPL) algorithm. Per each iteration, the JPL algorithm finds a non-maximal independent set of nodes, like the Jones-Plassmann algorithm. All nodes of the independent set are then colored with the same color, as per Luby's algorithm. Figure 3.4 shows an example of a graph colored following the JPL algorithm. The initialization phase populates the array of random numbers ρ (a). Each node is shown containing its corresponding random number. Nodes with the maximum random number in their neighbors are selected at the start of every loop, shown with a red border (b), (d), (f); only non-colored nodes are considered in these steps. After selecting the independent set, each node is colored with the same color (c), (e), (g). A different color is used in every iteration. A parallel algorithm that selects an independent set based on ordered numbers, such as the Monte Carlo rule by Jones and Plassmann, can be easily expanded to select two disjoint independent sets per iteration. While selecting nodes to insert in I so that their random number from the array of random



(a) Initialization (b) Loop 1 - Select IS (c) Loop 1 - Color (d) Loop 2 - Select IS

(e) Loop 2 - Color (f) Loop 3 - Select IS (g) Loop 3 - Color

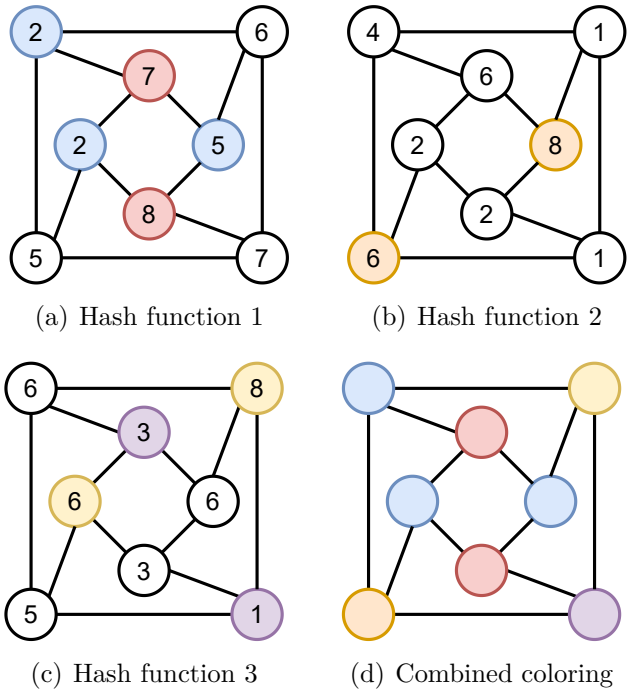
Figure 3.4. Jones-Plassmann-Luby algorithm applied to a simple graph.

values ρ is the maximum between their neighbors, it is also possible to find the independent set I_m where the random number is *minimum*. The two independent sets are disjoint, as a maximum number cannot be a minimum number at the same time (except for nodes with no edges, in which case the node is only inserted in one set). Cohen [15] proves that it is not possible to find more than two disjoint independent sets with this approach. The coloring is performed by assigning one color to I and a different color to I_m. By finding and coloring two independent sets per iteration, the processing time to color the whole graph is theoretically halved.

3.7 Cohen-Castonguay algorithm

Cohen and Castonguay [16] describe a method to fundamentally improve the JPL algorithm. Their idea is to replace the array of random values ρ with a hash function $H : V \to \mathbb{R}$. Using a function to generate random numbers instead of an array is a tradeoff between computation and memory bandwidth, the latter being a very limiting resource on modern GPU devices. Applying H to a vertex $v \in V$ produces a seemingly random number $H(v)$, that can be used in the same fashion as $\rho(v)$ to find two independent sets of local maximums and local minimums. Cohen and Castonguay push the idea further by being able to generate more than two disjoint independent sets in one iteration. As proven by Cohen [15], the method described

so far only allows for a maximum of two disjoint independent sets. The solution would be to employ more than one source of random numbers. By utilizing q hash functions $H_1, H_2, \ldots H_q$, the number of independent sets that can be generated is $2q$. However, these independent sets are not disjoint; it is possible that for node v, $H_i(v)$ is a local maximum or minimum, while also $H_j(v)$ is a local maximum or minimum. Since a node cannot have more than one color, some action must be taken before applying the coloring. Unfortunately, Cohen and Castonguay do not state how they combine the independent sets to obtain a valid coloring. A possible way would be to rank the hash functions, with H_1 being the most important and H_q the least important. With this method, a hash function H_i searches for two independent sets between the vertices that are not in any independent set found by higher ranking hash functions $H_j, j < i$. The practice of using more than one source of random numbers is generally unfeasible with arrays because they would occupy too much memory than the one available on a modern GPU. Figure 3.5 shows how



(a) Hash function 1 (b) Hash function 2

(c) Hash function 3 (d) Combined coloring

Figure 3.5. Cohen-Castonguay algorithm applied to a simple graph.

a simple graph can be colored in a single iteration using three hash functions as an example. In the example, the hashes produced by the functions are reported in each node (a), (b), (c). For the sake of simplicity, we use hash functions that output values in $[1, n]$ The independent sets of local maximums are highlighted with hot colors (red, orange, and yellow), while independent sets of local minimums are highlighted with cold colors (blue and purple). The independent sets are combined

to obtain a valid coloring (d).

The Cohen-Castonguay algorithm is available on CUDA in the cuSPARSE library through the CSRCOLOR routine. Some details of the implementation are hidden, like the number of hash functions q, or what type of hash functions are used. By running the algorithm and analyzing the coloring produced it is possible to estimate the value of q. Figure 3.6 shows, for every color, the number of nodes that have that color assigned after running the routine on a sample graph. From the plot, it is evident that the number of nodes colored with a certain color drastically decreases every 16 colors; from this, we can expect that $q = 8$, as eight hash functions generate sixteen independent sets and colors every iteration.

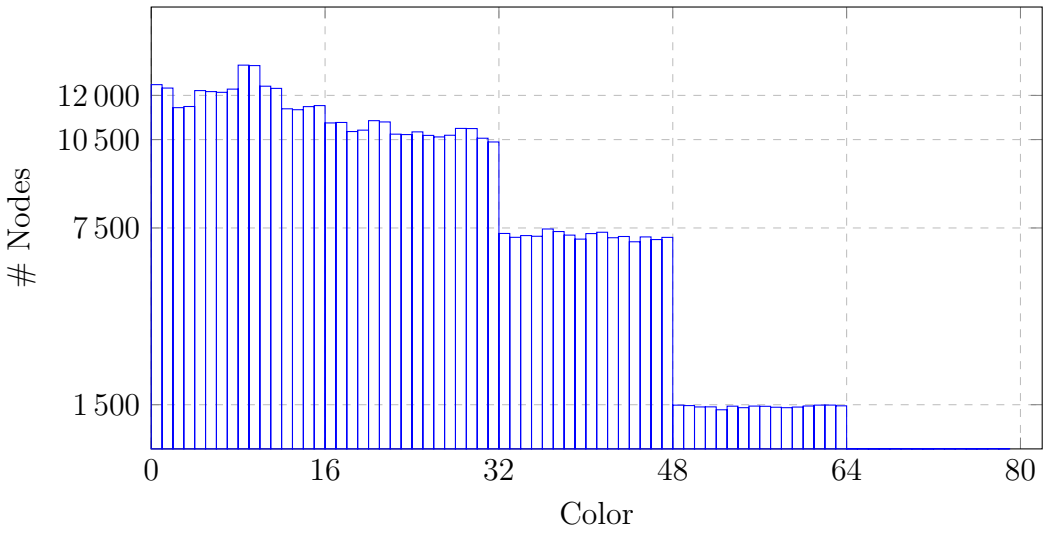


Figure 3.6. Number of nodes of each color for a run of the CSRCOLOR routine.

3.8 Jones-Plassmann algorithm

The Jones-Plassmann [14] algorithm is a parallel asynchronous algorithm to perform graph coloring. The main peculiarity of this algorithm is that it uses message passing as a means of communication and synchronization between the processors. The algorithm is formulated to use n processors, each coloring one vertex. The pseudo-code of the algorithm running on each processor is shown in Algorithm 7.

To perform the coloring, each processor needs to be aware of the colors of the vertices neighboring its node. This is easily achievable with message passing; a processor that has just colored its node will send a message to the processors of the node's neighbors, indicating the color used (line 19). The receiving processors will simply receive the color and store it until it is their time to assign a color (line 14). A receiving processor does not need to know *which* neighbor sent the

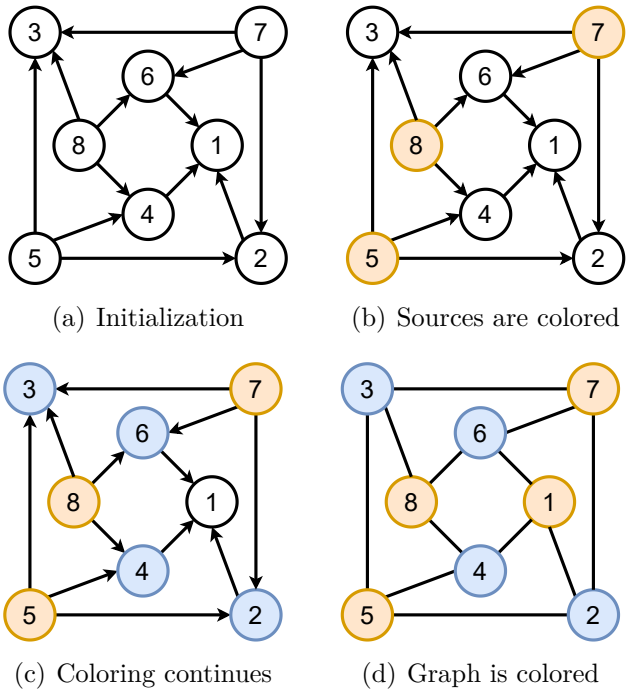
Algorithm 7 Jones-Plassmann algorithm

```
JONES-PLASSMANN (v, N(v))
 1: ρ(v) ← random number
 2: W ← ∅                          ▷ Set of nodes ∈ N(v) to be colored before v
 3: S ← ∅                          ▷ Set of nodes ∈ N(v) to be colored after v
 4: for w ∈ N(v) do
 5:     Send ρ(v) to processor responsible for w
 6:     Receive ρ(w)
 7:     if ρ(v) ≥ ρ(w) then
 8:         S ← S ∪ {w}
 9:     else
10:         W ← W ∪ {w}
11:     end if
12: end for
13: while W ≠ ∅ do
14:     Receive color(w)
15:     W ← W \ {w}
16: end while
17: color(v) ← smallest color not received from a neighbor of v
18: for w ∈ S do
19:     Send color(v) to processor responsible for w
20: end for
```

color, as the algorithm only needs the information that a neighbor is colored with the color received. The algorithm also establishes a system to allow some processors to perform the coloring, while the other processors wait to be allowed. This system is also achieved with message passing. At the start of the algorithm, each processor associated with a node $v \in V$ generates a random number $\rho(v)$ (line 1). It then sends $\rho(v)$ to the processors tasked to color the neighbors of v, $w \in N(v)$ (line 5), and receives from them $\rho(w)$ (line 6). Upon receiving $\rho(w)$, the processor immediately compares it with its random number. If $\rho(v) \geq \rho(w)$, vertex v will be colored before vertex w. To ensure this, vertex w is stored in the set S of vertexes to be colored after v (line 8). The processor will assign a color to v, then send the color used to all processors responsible for the neighboring nodes whose $\rho(w)$ was lower than $\rho(v)$, which shall wait for the message before starting its coloring. Otherwise, if $\rho(v) < \rho(w)$, the processor will increment a counter stating how many colors from its neighbors need to be received before it can start to color its vertex (line 10). The *receive* instructions on lines 6 and 14 are considered asynchronous, but need to be executed before exiting the loop surrounding them.

The effect of using this algorithm is analogous to extrapolating a DAG from the graph G. The DAG defines the order in which the nodes can be colored, from sources to sinks. The sources are the nodes that generate a $rho(v)$ greater than their neighbors, while sinks have the lower $rho(v)$ among their neighbors.

All nodes in the DAG have in-degree equal to the number of adjacent nodes $w \in N(v)$ that generated a random number $\rho(w) \geq \rho(v)$, and out-degree equal to the number of adjacent nodes that generated a random number $\rho(w) < \rho(v)$. Figure 3.7 shows how a small graph can be colored with the Jones-Plassmann algorithm. Numbers within each node represent the random number generated by that node. Edges are decorated with arrows pointing to the node with the lower random number between the pair, emphasizing the equivalence of coloring following a DAG. Random numbers are generated and exchanged between neighbors (a). Nodes receive information about the color chosen by their neighbors with a greater random number, before choosing a color of their own (b), (c), (d).



(a) Initialization (b) Sources are colored

(c) Coloring continues (d) Graph is colored

Figure 3.7. Jones-Plassmann algorithm applied to a simple graph.

Jones and Plassmann also propose a second algorithm, for distributed memory computing. Vertices are divided partitioned across p processors in blocks $\{V_i, \ldots V_p\}$. Each processor first divides its block into a set of local nodes V_i^L and a set of global nodes V_i^S. For block V_i, a global node shares at least one edge with another node outside of V_i; local nodes only share edges with other nodes inside V_i. V_i^S is colored first using the asynchronous heuristic from Algorithm 7, then V_i^L is colored sequentially with the greedy IDO heuristic.

Chapter 4

The Software

To study the multiple coloring algorithms presented in Chapter 3.3, we write a piece of software, with the object of collecting data regarding runtime and the number of colors used. The program produces a suite of executables, compiled from the same source code. We take advantage of class inheritance of C++ and conditional compilation via pre-compiler directives to produce many executables, each implementing a different algorithm and heuristic. Despite each implementing a different algorithm, all programs follow the same macro operations. First, the program parses the command line arguments, and the graph is loaded from file to main memory. Then the graph is colored with the coloring algorithm. After checking that the coloring produced is a valid solution, the program prints data about the elapsed time for the computation and the number of colors used. In this chapter, we provide details on the software and implementation of the algorithms. In Chapter 5.1 we describe modifications we apply to the code based on experimental evidence.

Command line arguments contain information on how to customize the program execution and the data printed after the computation. The path to the file containing the graph to color always needs to be specified for the program to continue execution. Four optional flags can also be provided as command arguments:

- `-h`: prints the user guide and terminates.
- `-r`: must be followed by a positive integer number, indicating how many times the algorithm should be run. When specified, the coloring is performed on the graph multiple times, with a reset of the starting conditions before each run, The statistics about elapsed time and colors used are stored in appropriate data structures, and printed at the end of the execution; the program also computes and prints the average of the values. Using this flag, the graph is loaded into main memory only once at the start of the execution.
- `-c`: prints the pairs vertex-color at the end of the output. If the flag is followed by the path to a file, the pairs are written to that file instead. If used with `-r`,

only the pairs of the last coloring are printed.

- `-H`: prints the histogram of colors at the end of the output. The histogram is a list of pairs of color-number of vertices colored with that color. If the flag is followed by the path to a file, the histogram is written to that file instead. If used with `-r`, only the histogram of the last coloring is printed.

Any argument not recognized by the program is automatically skipped, printing a warning message,

The program reads the graph to color from a file stored on disk. The path to the file is passed as a command line argument. The graph is stored as adjacency lists in textual form: the first line contains a single number n that indicates how many lines follow. Each following line represents a single node, listing the edges connecting it to the other nodes. Lines start with the index of the node in the range $[0, n-1]$ followed by a `:` character, and end with the `#` character. In between, we expect a space-separated list of the nodes that share an edge with the current node.

The representation of the graph in main memory determines which input algorithms are available We implement multiple input algorithms to read and parse the file. Selecting the graph representation as well as the input algorithm to use is done at compile time by defining the correct constants to perform conditional computation. The constants to select which data structure we use for graph representation are `GRAPH_REPRESENTATION_ADJ_LIST` and `GRAPH_REPRESENTATION_CSR`, which respectively activate the adjacency lists and compressed sparse row format. The compressed sparse row format allows only for a sequential scan of the input; no constants need to be defined, as this is the only supported algorithm. The adjacency lists format defines three methods to read the input file. The methods are activated by defining one of the following constants:

- `SEQUENTIAL_INPUT_LOAD`: one thread reads one line of the input file and populates the adjacency list, looping until the end of the file.
- `PARALLEL_INPUT_LOAD`: multiple threads work concurrently towards parsing the file. Each thread follows the same algorithm as the sequential input version. The lines are provided to the threads by a queue-like object, protected from concurrent access by a mutex.
- `PARTITIONED_INPUT_LOAD`: the file is completely loaded in main memory, and divided into partitions of the same size. Each thread is assigned a partition. Because partitions do not start or end at the end of a line, as each line has an arbitrary length, threads will align their start and end positions to the next newline character. The parsing then proceeds in the same way as the other versions.

We tried to develop a fourth method to read the input file in parallel: similar to partitioned load, we divide the input file into multiple blocks. The difference stands

in the input file, which we preemptively modified with a Python script [19]. The modified file encodes information in binary, and the adjacency lists of nodes are padded so that all lists have the same length. In this way, we can make partitions of precise dimensions, and we do not need to align the partitions to the next newline character. However, this method is not included in the software, because the dimensions of a graph file in binary format are around 500 times larger than in textual format. The massive increase in dimension is caused by the padding. In fact, the graphs we use for testing, shown in Table 5.1, are sparse; adding padding to make all adjacency lists equal in length causes the file to contain n adjacency lists of Δ elements. In cases where $\Delta \gg \overline{\delta}$ the file size increases a lot, as most of the adjacency lists are filled with padding. Moreover, reading and parsing a binary file this large takes around 1000 times more than using the standard partitioning method. We deem this method of using partitioning on a binary version of the graph unfeasible, and discard it from the implementation of the software. Via testing, we find that the partitioned load is the best-performing input reading algorithm for the adjacency lists format, reaching a loading speed around 3 times faster than the sequential load method. Not surprisingly, the parallel load method is the worst in terms of speed out of the three; this is likely due to thread synchronization, as we use a mutex to protect concurrent reading from the input file. A likely solution to speed up the process would be to load the file completely beforehand like we already do with the partition method. Despite the fast partitions-based input method available with adjacency lists, we decide to mainly use the CSR format in our tests that only provides sequential load. CSR is the superior format when it comes to being transferred to GPU memory since it requires only two vectors to be moved. Moreover, we notice an average of 20% speed increase on the CPU by using CSR instead of adjacency lists, likely due to cache permanence. As we do not want to poison our test results by changing the graph format between algorithms, we decide to only use CSR.

Time benchmarks are operated by the Benchmark class. The class manages several Benchmark objects, one per each run specified with the flag `-r`. Each run of the algorithm is treated independently by interacting with a unique Benchmark object. A benchmark object can be retrieved from every part of the program with the static method `getInstance(int)`; the integer parameter is used to determine which instance to return. A single Benchmark object holds multiple time interval durations in a map structure, indexed with an integer key. This allows a single object to be used to benchmark many stages of the program execution by storing multiple durations. The method `sampleTime()` is used at the start of a block of code that we want to benchmark. At the end of the benchmark process, we call the method `sampleTimeToFlag(int)`: the integer parameter is used as a key to the map of durations to select which time counter should be incremented. The counter is incremented by the difference between the current time and the last time `sampleTime()` has been called. The method `sampleTimeToFlag(int)` works by

calling `sampleTime()`, so multiple calls of `sampleTimeToFlag(int)` can be chained with no `sampleTime()` calls in between. In the software, we use flag 0 to hold the time to load and read the file from disk, flag 1 to hold the time to perform the preprocessing stage, where we allocate and initialize the objects related to the graph coloring algorithm used. The time spent coloring the graph is stored under flag 2, and flag 3 is used for the postprocessing step, where held resources are freed. The Benchmark class exposes static methods to compute the average of the durations of the same flag throughout all benchmark objects.

The program displays information to the user by printing lines in the terminal. First, the information about the program is printed; these include the version number, graph representation, and coloring algorithm chosen at compile time. The file-loading phase starts with the output of a notification message. When the loading has finished, a success message is printed, along with data about the graph. The coloring phase is next. After each run of the algorithm, a line detailing the computation is printed; the line reports the number of colors used and the coloring time. Each line in this phase is separated by a separator line. After all the runs are concluded, the program prints statistics about the computation. All time measures are reported in seconds. The output of a sample run is presented as an example in Figure 4.1.

4.1 Implementation details

The part of the software that deals with storing the graph in memory is modeled as a class hierarchy. The AdjacencyLists and CompressedSparseRow classes define the details of how data is stored. To hide the details we provide a common API through the GraphRepresentation abstract class. The class hierarchy for graph representation is shown in Figure 4.2 as a UML diagram. Child classes define their way of storing and accessing graph data, and override the abstract definition of three methods to allow the rest of the program to interact with the graph. The method `exists_edge(int,int)` returns `true` if nodes v and w are connected by an edge. The pair of methods `begin_neighs(int)` and `end_neighs(int)` provides a way of exploring $N(v)$ with the use of iterators. The last method, `count_neighs(int)`, returns the cardinality of the neighbors set $N(v)$; the method is not abstract, as it is implemented by calling `begin_neighs(int)` and `end_neighs(int)`, exploiting the concept of polymorphism provided by C++.

The software gives access to five algorithms that perform graph coloring. The algorithm is chosen at compile time by defining a constant for the pre-processor. Each algorithm can be further controlled with other constants, to enable different behaviors of the algorithm. The constants are:

- `COLORING_ALGORITHM_GREEDY`: enables coloring with the sequential greedy algorithm. The algorithm is implemented for CPU architectures. The algorithm

```
$ ./greedy_ldf ./email_enron.gra -r 5
Graph Coloring - v2022.9.26
Graph Representation: Compressed Sparse Row
Coloring Algorithm: Greedy Largest Degree First

Loading graph from ./email_enron.gra
Graph successfully loaded from file.
n: 36692, m: 367662, maxD: 1383, minD: 1, avgD: 10.0202
=========================================================
1       num-cols: 29    time: 0.011383
=========================================================
2       num-cols: 29    time: 0.010311
=========================================================
3       num-cols: 29    time: 0.011142
=========================================================
4       num-cols: 29    time: 0.009587
=========================================================
5       num-cols: 29    time: 0.011678
=========================================================
Load time: 0.124413
Avg pre-process time: 9.58e-05
Avg process time: 0.0108202
Avg post-process time: 0
Avg total time: 0.135329
Avg number colors: 29
```

Figure 4.1. Sample output

can be customized by changing the policy for choosing the next node to color with the following constants:

  - `SORT_LARGEST_DEGREE_FIRST`: Largest Degree First ordering. Nodes with a higher degree are colored first.
  - `SORT_SMALLEST_DEGREE_FIRST`: Smallest Degree First ordering. The reverse ordering of Largest degree First. Nodes with a higher degree are colored last.
  - `SORT_VERTEX_ORDER`: Lexicographical ordering, also called First Fit (FF). Nodes are colored in order of ascending index.
  - `SORT_VERTEX_ORDER_REVERSED`: Reversed Lexicographical ordering. Nodes are colored in order of descending index.

- `COLORING_ALGORITHM_GM`: enables coloring with the Gebremedhin-Manne algorithm. The algorithm is implemented for multi-core CPU architectures. The policies of the algorithm can be controlled with the following constants:

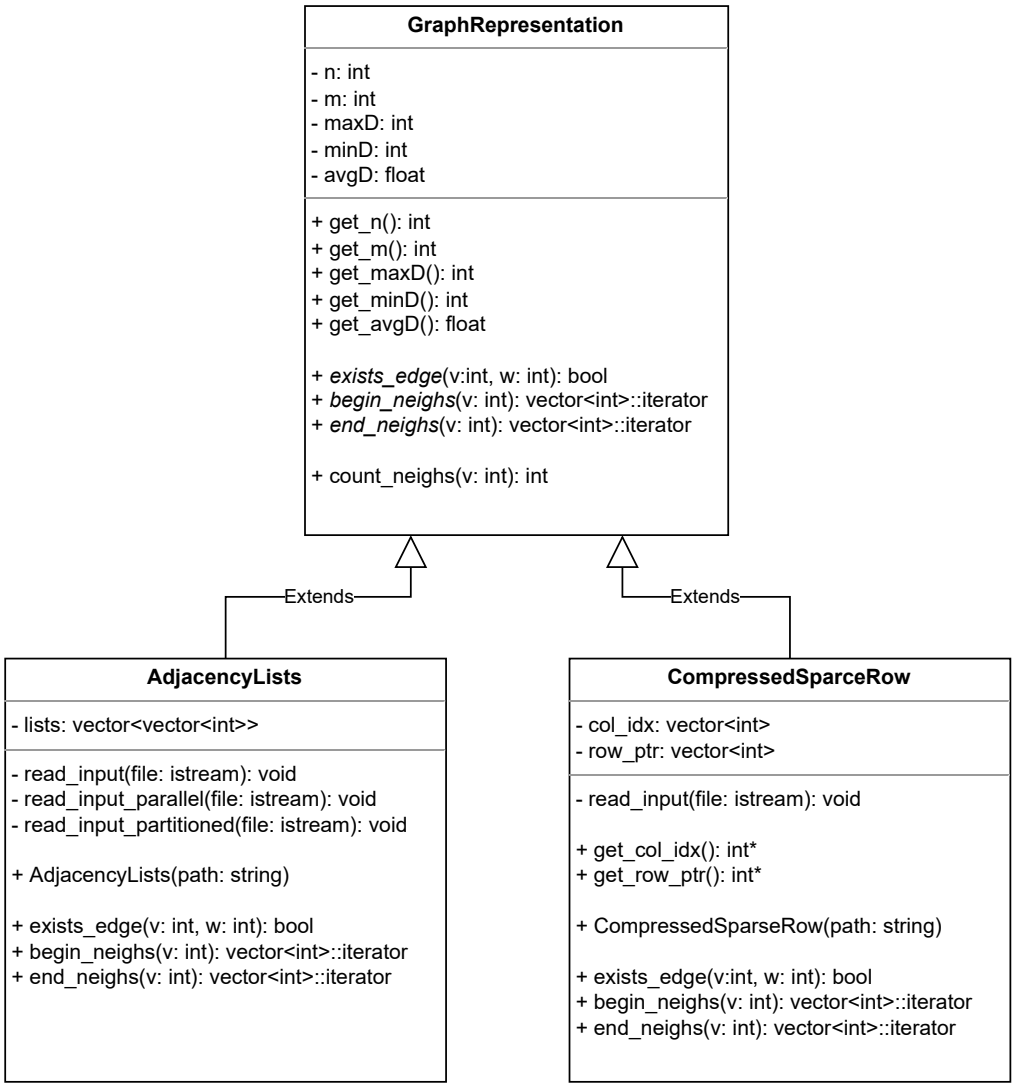


Figure 4.2. Graph representation UML diagram

  - `COLORING_SYNCHRONOUS`: enables synchronization between threads. Synchronization is achieved with the use of barrier primitives.
  - `COLORING_ASYNCHRONOUS`: disables synchronization between threads.
  - `USE_STANDARD_ALGORITHM`: enables the use of the standard version of the algorithm.
  - `USE_IMPROVED_ALGORITHM`: enables the use of the improved version of the algorithm.
- `COLORING_ALGORITHM_JP`: enables coloring with the Jones-Plassmann algorithm. The implementation presents the following differences from the original

formulation. We partition the vertex set V in blocks, one per each working thread. We also simulate message passing by storing the message contents in global memory. The algorithm is implemented for multi-core CPU architectures. The policy for partitioning vertices can be specified with:

  - `PARTITION_VERTICES_EQUALLY`: vertices are split into partitions so that each partition contains a similar number of vertices.
  - `PARTITION_VERTICES_BY_EDGE_NUM`: vertices are split into partitions so that each partition contains a similar number of edges.

- `COLORING_ALGORITHM_JPL`: enables coloring with the Jones-Plassmann-Luby algorithm. The algorithm is implemented for many-core GPU architectures using the CUDA framework. The policy on the number of independent sets per iteration can be set with:

  - `COLOR_MAX_INDEPENDENT_SET`: finds and colors the independent set of maximums every iteration.
  - `COLOR_MIN_MAX_INDEPENDENT_SET`: finds and colors the independent sets of maximums and minimums every iteration.

- `COLORING_ALGORITHM_CUSPARSE`: enables coloring with the CSRCOLOR routine from the cuSPARSE library. The program offers a wrapper around the routine, but cannot modify its behavior.

Each algorithm is modeled as a class extending the ColoringAlgorithm abstract class. The class structure of coloring algorithms is shown in Figure 4.3 as a UML diagram. The API provided by the ColoringAlgorithm abstract class consists of three public methods available to the external program and one protected method that should be used by the inheriting classes. Methods `init()` and `reset()` are called to initialize the object at the start of the execution; `reset()` is also called in-between multiple runs of the algorithm to restore the data structures to their original state. The method `start_coloring()` instructs the program that everything is ready to execute the coloring algorithm. After the execution, it returns the number of colors used by the algorithm to color the graph. Method `suggest_vertex_color(int)` receives the index of a vertex as input and returns a suitable color that could be assigned to that node as output. The color is chosen by iterating over all neighboring nodes, marking off in a vector which colors are already used. The color returned is the smallest non-marked color.

In the rest of this chapter, we present each algorithm in depth, analyzing the UML diagram from Figure 4.3 and showing some snippets of the code. We start using the naming convention reported in Table 4.1 in preparation for Chapter 5, where we compare the many variations of the algorithms that have been implemented in the software.

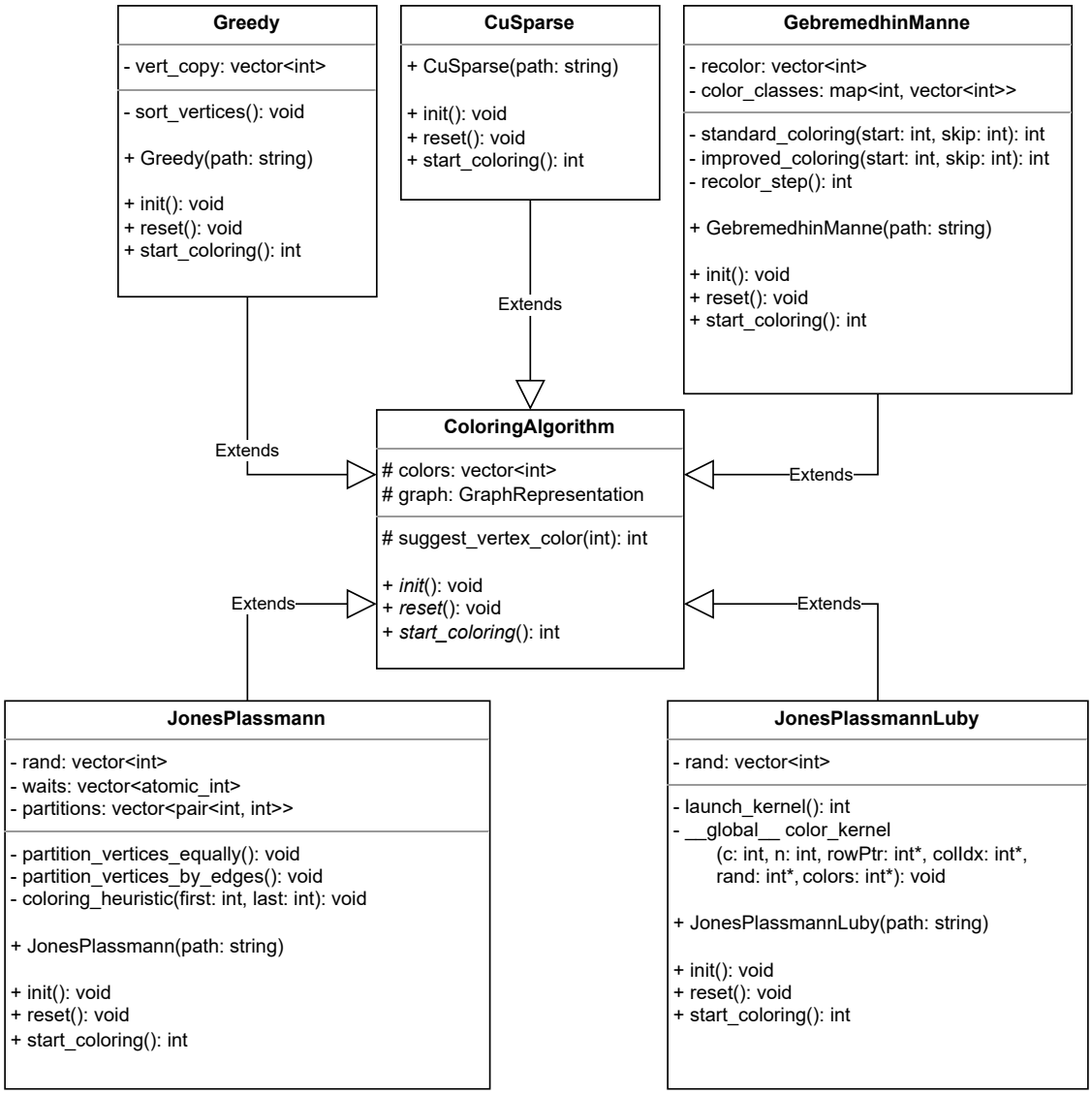


Figure 4.3. Coloring algorithm UML diagram

4.2 Method A - Greedy

The Greedy class implements the greedy algorithm described in Chapter 3.4. Contrary to how the algorithm has been described, our implementation requires that the order in which the nodes are colored is chosen before the first color is assigned. As such, Incidence Degree and Saturation Degree orderings are not implementable with this class, as both orderings change dynamically after a node is colored. Instead, we

Method		*Algorithm*	*Architecture*
A		Greedy	
	A1	Greedy First Fit	Sequential CPU
	A2	Greedy Reversed First Fit	Sequential CPU
	A3	Greedy Largest Degree First	Sequential CPU
	A4	Greedy Reversed Largest Degree First	Sequential CPU
B		Gebremedhin-Manne	
	B1	Gebremedhin-Manne Synchronous Standard	Multi-core CPU
	B2	Gebremedhin-Manne Synchronous Improved	Multi-core CPU
	B3	Gebremedhin-Manne Asynchronous Standard	Multi-core CPU
	B4	Gebremedhin-Manne Asynchronous Improved	Multi-core CPU
C		Jones-Plassmann	
	C1	Jones-Plassmann with Vertex Partition	Multi-core CPU
	C2	Jones-Plassmann with Edge Partition	Multi-core CPU
D		Jones-Plassmann-Luby	
	D1	Jones-Plassmann-Luby Max Independent Set	Many-core GPU
	D2	Jones-Plassmann-Luby Min Max Independent Set	Many-core GPU
E		Cohen-Castonguay (cuSPARSE implementation)	Many-core GPU
F		Jones-Plassmann-Luby (Gunrock implementation)	Many-core GPU

Table 4.1. Naming convention for algorithms

provide First Fit and Largest Degree First orderings, and their reversed counterparts. As shown in the UML diagram of Figure 4.3, the class contains a private vector of integers `vert_copy`; this is the vector that contains the nodes in the order they will be colored in. The vector is assigned numbers from 0 to $n-1$ during `reset()`, and its order is changed with the private method `sort_vertices()`, which is called at the start of the overridden method `start_coloring()`. `sort_vertices()` calls the `sort` function from the standard library to alter the order of `vert_copy`. The ordering is achieved by using the pre-compiler to assign the correct compare lambda function to a compare variable used on the `sort` function, depending on the selected ordering. The execution of `start_coloring()` continues by iterating over the vertex indexes in `vert_copy`, now properly ordered, assigning to each one the return value of the method `suggest_vertex_color(int)` when the node is passed as a parameter. Snippets of the code are shown in Figure 4.4.

4.3 Method B - Gebremedhin-Manne

The GebremedhinManne class implements four modes —listed in Table 4.1—of the Gebremedhin-Manne algorithm described in Chapter 3.5. From the UML diagram

```
void Greedy::sort_vertices() {
    // Define lambdas
    auto sort_ff =
        [&](const int v, const int w) { return v < w; };
    auto sort_rff =
        [&](const int v, const int w) { return v > w; };
    auto sort_ldf = [&](const int v, const int w) {
        return this->graph.count_neighs(v) >
            this->graph.count_neighs(w);
    };
    auto sort_rldf = [&](const int v, const int w) {
        return this->graph.count_neighs(v) <
            this->graph.count_neighs(w);
    };

    // Assign correct lambda
    auto sort_lambda =
#if defined(SORT_VERTEX_ORDER)
        sort_ff;
#elif defined(SORT_VERTEX_ORDER_REVERSED)
        sort_rff;
#elif defined(SORT_LARGEST_DEGREE_FIRST)
        sort_ldf;
#elif defined(SORT_SMALLEST_DEGREE_FIRST)
        sort_rldf;
#endif

    // Perform sort
    std::sort(
        this->vert_copy.begin(),
        this->vert_copy.end(),
        sort_lambda
    );
}
```

Figure 4.4. Nodes get sorted in the Greedy class

in Figure 4.3, the class contains a private vector of integers `recolor`, populated during the conflict search step with the indexes of nodes that cause a conflict and need to be recolored. It also contains the private map `color_classes`; colors are used as keys to access a vector containing all vertices with that color. The map is only used in the improved versions of the algorithm to determine the order of the coloring. As the algorithm is formulated as parallel, we use p threads, where p is equal to the number of cores of the CPU running the program, to lower the effect of the overhead caused by context switching. The `init()` method is overridden to reserve memory space for the `recolor` vector; we use

the formula $n\,(p-1)\,\overline{\delta}/2\,(n-1)$, proposed in the original paper [12], to estimate the maximum possible number of conflicts, and avoid array reallocations during the processing. The `start_coloring()` method starts the threads, waits for their completions, and executes the `recolor_step()` method. The threads execute either the `standard_coloring(int,int)` or `improved_coloring(int,int)` method, depending on the selected mode. Both methods accept two integers as input parameters. The first parameter indicates the starting index for that thread's computation and can vary between 0 and $p-1$. The second parameter indicates how many nodes to skip to find the next node to color, and it is set to p for all threads. As an example, we analyze thread p_i; the thread receives i, p as input parameters, and analyzes all nodes in $\{v \mid v \bmod p = i,\, v \in V = \{0,1,\ldots,n-1\}\}$ In this way, we dynamically create blocks based on the node's lexicographical position, without needing additional memory or computation. For synchronous modes of the algorithm, however, we add $p-(n \bmod p)$ to be considered. We call these extra nodes *ghost* nodes because they are needed to keep threads busy, but are not colored as they do not exist. We need ghost nodes so that threads that are assigned blocks with one less node than the others do not skip the last iteration of the for loop during the pseudo-coloring step. At the end of the for loop we perform a barrier synchronization instruction; this barrier is the reason the algorithm is synchronous. The barrier is initialized to stop threads execution until p threads are waiting on the barrier. If some threads exit the for loop before the last iteration, the other threads remain stopped on the barrier, without any possibility of continuing execution. With ghost nodes, we keep threads that would terminate early on the for loop for one more iteration, to meet the number of expected threads on the barrier. This is necessary when n/p is not an integer, and nodes cannot be split evenly across all blocks. The asynchronous modes of the algorithm do not require the use of ghost nodes, as the barrier synchronization instruction is not present. A code snippet of the iteration is reported in Figure 4.5.

After checking that the considered node is not a ghost node, threads perform coloring in parallel. Coloring is done by selecting the smallest color not used by the neighbors of the node. For synchronous modes, the original formulation explains that, instead of considering all the neighbors $N\,(v)$ to color node v, we can reduce the number of checks by intersecting $N\,(v)$ with the set of colored nodes up to that point. Since the algorithm is synchronized with a barrier, there is no risk of race conditions. We decide to not implement it this way, and prioritize code reuse by calling the method `suggest_vertex_color(int)`, provided by the ColoringAlgorithm class. In the method `improved_coloring(int,int)` this color is also used as an index to insert the vertex in the vector of the `color_classes` map. This insertion is protected by a lock on a mutex shared by all threads.

The `improved_coloring(int,int)` method continues its execution by performing the second pseudo-coloring step, detailed in Algorithm 3. We do not use two different arrays as detailed in the algorithm, so the first step is to reset the values

```
void GebremedhinManne::standard_coloring(int start, int skip) {
    int n = this->graph.get_n();
    // Step 1 - pseudo-coloring
#if defined(COLORING_ASYNCHRONOUS)
    for (int v = start; v < n; v += skip) {
        this->colors[v] = this->suggest_vertex_color(v);
    }
#elif defined(COLORING_SYNCHRONOUS)
    int nCeil = (n / skip + 1) * skip;
    for (int v = start; v < nCeil; v += skip) {
        if (v < n) {
            this->colors[v] = this->suggest_vertex_color(v);
        }
        barrier->wait();
    }
#endif
    ...
}
```

Figure 4.5. Implementation of the pseudo-coloring step for Methods B1 and B3

contained in the `colors` vector to a value that indicates "no color". Threads partition the work to reset the `colors` vector the same way as for the pseudo-coloring step; ghost nodes are not needed because iterations are not synchronous during this reset stage. We iterate in reverse order over the color classes, starting from the largest color and ending at 0. Each color class is divided into blocks, in the same way explained for the first pseudo-coloring step, so that each thread works independently on its nodes with the method `suggest_vertex_color(int)`. For this process, Gebremedhin and Manne do not explain how synchronization should be achieved. We decide to synchronize with a barrier after completely coloring each color class, for the synchronous mode only. The ghost nodes technique is not used in this case. Another possible solution would be to synchronize the coloring of each node; ghost nodes would be needed in this case.

Both the `standard_coloring(int)` and `improved_coloring(int)` methods perform the conflict search step after coloring. Each thread checks the nodes on its assigned block; for each node, its color is compared against the color of its neighbors. If they have the same color, the one with the lowest index is uncolored and saved in the `recolor` vector. In this step, we skip all neighboring nodes that were not colored during the same time frame as the node currently considered. This is possible only in Method B1, as time frames have no meaning in the asynchronous modes, and lose meaning during the second pseudo-coloring for the improved mode.

When all threads terminate, the main thread resumes the execution inside the `start_coloring()` method. The `recolor_step()` method performs recoloring by

iteratively calling `suggest_vertex_color(int)` on every node in the `recolor` array. The coloring terminates by returning the number of colors used, obtained with a linear scan of the `colors` vector searching for the maximum value, as colors are assigned in order.

4.4 Method C - Jones-Plassmann

The JonesPlassmann class implements a variation of the algorithm by Jones and Plassmann presented in Chapter 3.8. While the original formulation requires an architecture that can allocate and execute a processor per node of the graph, using message passing to achieve inter-processor communication, we opt for a more classical approach, with few threads that color each a partition of V, and global memory as a mean to exchange information. Not following the original formulation was a mistake committed at the start of development, caused by ingenuity on our part. We extend the base class with three private vectors, as seen in the UML diagram of Figure 4.3. Vector `rand` contains integers, vector `waits` contains atomic integers, and vector `partitions` contains a pair of integers. We use atomics because the vector `waits` is accessed by multiple threads in a pattern that makes race conditions possible.

We override the `init()` method to populate the `partitions` vector by calling either the `partition_vertices_equally()` or `partition_vertices_by_edges()` private method, depending on the option selected at compile time. The `partitions` vector contains pairs of first and last nodes so that all nodes in between the two are part of the partition. The `partition_vertices_equally()` method creates partitions of $\lfloor n/p \rfloor$ sequential nodes, where p is the number of threads. The `partition_vertices_by_edges()` method is a bit more convoluted. It first computes the number of edges a partition should contain to have them divided equally among partitions, as the threshold $t = \lfloor m/p \rfloor$. Then it iteratively creates p partitions by adding one node at a time, checking that the number of edges connected to the nodes in the partition is lower than t. When the threshold is surpassed, it repeats the process for the next partition. Partitions have a similar number of nodes when `partition_vertices_equally()` is used, and have a similar number of edges when `partition_vertices_by_edges()`. By computing them in the `init()` method, partitions remain the same across multiple executions.

In the `reset()` function we initialize the `waits` vector and populate the `rand` vector. We initialize `waits` with zeroes, and `rand` so that each cell contains its index, from 0 to $n - 1$. Then we call the `shuffle` function from the C++ standard library to get a random permutation of the `rand` array. We do the permutation instead of a more canonical random number generation to avoid dealing with adjacent nodes with the same random number.

The `start_coloring()` method is overridden to spawn p threads. We decide to

use p threads, so that p is equal to the number of cores of the CPU running the program, to lower the effect of the overhead caused by context switching. Threads execute the `coloring_heuristic(int,int)` method. The parameters of the method are the first included and last excluded vertices in the partition assigned to the thread, retrieved from the `partitions` array. Parallel execution starts by populating the `waits` array. Each thread loops on the nodes in the partition, and saves in `waits` how many adjacent nodes have a random number greater than the one belonging to the current node. Values of the `waits` array are interpreted as timers that need to reach 0 before the corresponding node can be colored. This preparation step is followed by a barrier synchronization, to make sure all threads are finished counting. Threads then start looping on nodes of their partition to assign colors; the code is shown in Figure 4.6. In the loop, we color nodes when their

```
void JonesPlassmann::coloring_heuristic(int first, int last) {
    ...
    bool again = true;
    int const n = this->graph.get_n();
    do {
        again = false;
        for (int v = first; v < n && v < last; ++v) {
            if (this->waits[v] == 0) {
                // Color current node v
                this->colors[v] = this->suggest_vertex_color(v);
                // Decrease v's timer under 0 to
                //  avoid multiple recolors
                --this->waits[v];
                // Decrease v's neighbors timer
                auto const end = this->graph.end_neighs(v);
                for (auto it = this->graph.begin_neighs(v);
                    it != end; ++it)
                {
                    --this->waits[*it];
                }
            } else if (this->waits[v] > 0) {
                again = true;
            }
        }
    } while (again);
}
```

Figure 4.6. Implementation of the coloring in the Jones-Plassmann algorithm

timer in the `waits` array reaches 0. The color assigned when the timer reaches 0 is equal to the return of the `suggest_vertex_color(int)`. Right after coloring node v, we subtract one from the timers of nodes neighbors to v. Threads terminate when all nodes of their respective partition are colored. The coloring terminates

by returning the number of colors used, obtained with a linear scan of the `colors` vector searching for the maximum value, as colors are assigned in order.

4.5 Method D - Jones-Plassmann-Luby

The JonesPlassmannLuby class implements the Jones-Plassmann-Luby algorithm explained in Chapter 3.6. We implement the algorithm for many-core GPU architectures in CUDA. We can select how many independent sets are searched every iteration—one or two—at compile time via the correct pre-compiler definition. We expect that finding and coloring two independent sets can be twice as fast as only finding and coloring one, as the second set is found and colored in constant time with respect to the first.

Analyzing the UML diagram in Figure 4.3, the class contains a vector of integers called `rand`. Its purpose and initialization are the same as for the `rand` vector of the JonesPlassmann class presented in Chapter 4.4. In the `reset()` method, the vector gets filled with unique integer numbers from 0 to $n-1$ and gets immediately shuffled with the `shuffle` function to produce a random permutation. Each number is linked to a vertex through the index it is stored at, augmenting the vertex with a random number used to create independent sets. The `start_coloring()` method checks that the graph representation format chosen at runtime is CSR, otherwise the program is terminated with an error. If no errors arise, the execution continues by calling `launch_kernel()`.

The private method `launch_kernel()` acts as an interface between the host code and the device code by managing the device memory and launching the kernel function with appropriate parameters. The algorithm requires global synchronization after every iteration. We achieve it by designing the kernel code to execute a single iteration of the algorithm, launching the kernel multiple times; terminating the kernel execution, and continuing with the host code acts as an implicit global synchronization of the CUDA threads.

The `color_kernel(int,int,int*,int*,int*,int*)` method is qualified with the `__global__` keyword, and contains the device code. The first parameter is an integer that indicates the number of iterations completed so far; it is used to determine which colors to use. The second parameter is the cardinality n of the graph. The third and fourth parameters are integer pointers of the row pointers and column index arrays of the CSR representation of the graph. The fifth parameter is the integer pointer of the `rand` array, and the sixth parameter is the integer pointer of the `colors` array. The pointers act as the entry point to the arrays in the device's global memory. The pointers are obtained from the cudaMalloc API call, and the data is transferred with cudaMemcpy in the `launch_kernel()` method. The `colors` array, however, is treated differently. We use a `device_vector` from the Thrust library. Device vectors are objects similar to vectors from the C++ standard

library, with the exception that reside in the device's global memory. Thrust provides high-level APIs to manage device vectors in the GPU directly from the host code. A total of $4(3n + m + 1)$ bytes are needed in the device's global memory to house the data needed for coloring with methods D, assuming the use of 32 bit integers.

The `launch_kernel()` method sets up the GPU execution layout by calculating the dimension of the grid and its blocks. We define the block dimensions as a 1-dimensional array of 256 threads, and the grid dimensions as a 1-dimensional array that contains $\lceil n/256 \rceil$ blocks. In this way, launching the kernel spawns at least n threads, so that each one computes the color of one node. The extra $n \bmod 256$ threads that do not have a node assigned for coloring do not perform any computation. The kernel is launched inside a for loop. The loop helps us count how many iterations we are performing, and continues until all nodes are colored. After the kernel terminates, we check how many nodes are left to be colored. To do so we call the `count` function from the Thrust library on the `colors` device vector. The function counts how many elements between two iterator objects are equal to a given value; in our case, we use -1 to indicate a non-colored node. The counting is completely performed on the GPU, as we pass iterators obtained from a device vector. The algorithm loops, performing the kernel launch and the count operation, until all nodes are colored.

The `color_kernel(int,int,int*,int*,int*,int*)` method is the kernel containing the code to be executed on the GPU device. Using the nomenclature introduced in Table 4.1, the kernel definition for method D2 is reported in Figure 4.7. The definition for method D1 can be simply obtained by removing lines 10, 30 and 35, and initializing `color = c` on line 9. Each device thread checks if its assigned node $v \in V$ is already colored, in which case all the computation is skipped. If it is not colored, we check each neighboring node $w \in N(v)$ in a for loop. We skip v if it is present in the list of neighboring nodes because of a looping edge, and neighboring nodes that were colored in a previous iteration; however, we consider nodes that were colored in the current iteration. For each neighboring node we consider, we compare `rand[w]` with `rand[v]`. For method D1, we initially assume that v is part of the independent set of maximums; for method D2, we assume that v is part of both independent sets of maximums and minimums. Participation in a set is represented as a local boolean variable. By comparing the two random values, we decide if v remains in the sets or needs to be removed from one or both of them. The choice of removing v from an independent set is done without *if* statements, to avoid the creation of divergent branches. In case v is not part of any set, we do not break out of the for loop, to reduce the effect of divergence. After the iteration, we are ready to assign the colors. For method D1, we assign one color, equal to the first input parameter of the kernel function c. For method D2, we assign color $2c$ to the nodes in the set of maximums and color $2c + 1$ to the nodes in the set of minimums. In case a node is in both sets because it has

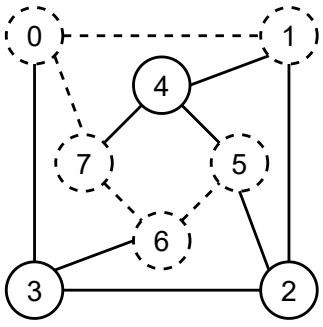
no uncolored neighbors, we always assign color $2c$. We tried to approach assigning colors in a branchless manner, but the code lost in readability, and it did not reflect a gain in performance.

```
__global__ void JonesPlassmannLuby::color_kernel(
    int c, int n, int* row_ptr,
    int* col_idx, int* rand, int* colors)
{
    for (int v = threadIdx.x + blockIdx.x * blockDim.x;
        i < n;
        i += blockDim.x * gridDim.x
    ) {
        int  color    = 2 * c;
        bool localmin = true;
        bool localmax = true;

        if (colors[v] != -1) continue; // Ignore colored nodes

        // Loop on neighbors
        for (int i = row_ptr[v]; i < row_ptr[v+1]; ++i) {
            int w  = col_idx[i];
            int cw = colors[w];

            int vr = rand[v];
            int wr = rand[w];

            if ((cw != -1 && // Ignore colored neighbors
                // Consider neighbors colored this iteration
                cw != color && cw != color+1) ||
                v == w) // Ignore looping edges
                continue;

            // Determine inclusion in independent sets
            localmin &= vr < wr;
            localmax &= vr > wr;
        }

        // Assign color
        if (localmin) colors[v] = color + 1;
        if (localmax) colors[v] = color;
    }
}
```

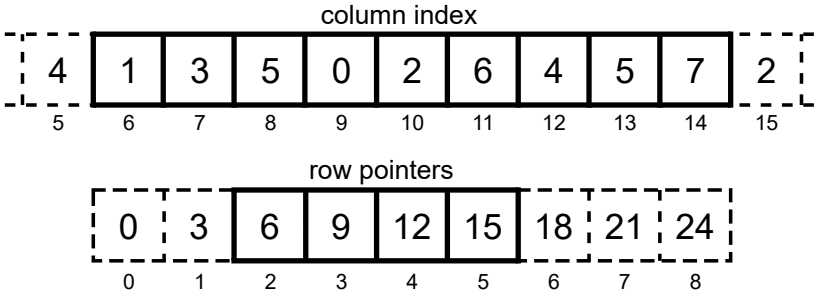
Figure 4.7. Kernel function for method D2

Given the limited memory available on GPU devices, we extend the implementation presented above by allowing a coloring to be produced by coloring multiple subgraphs. Our idea is to partition a graph that does not fit into the limited device

memory into subgraphs that are small enough to fit, and independently color each subgraph. Edges that lay between two subgraphs connect two nodes that can be colored with the same color, thus generating a conflict. When all subgraphs are colored we search for conflicting pairs of nodes and correct the conflict. Our first approach is to employ a graph partitioning algorithm to divide a graph into subgraphs. We use the hMETIS [20] library to perform the partition. However, we find that partitioning a small graph with 500 000 nodes and 17 500 000 edges, which normally occupies $4\,(3n + m + 1) \approx 76$ MB and can fit perfectly fine in modern GPUs memory, takes more than 1 minute with hMETIS, invalidating our fast coloring implementation. Our second approach is to create pseudo-partitions by adding nodes in lexicographical order in a subgraph. We call them "pseudo"-partitions because, by adding nodes, we also add the edges they are connected to, without checking that both ends of each edge are present in the pseudo-partition. The pseudo-partition is defined by selecting a starting and an ending index in the row pointer vector of the CSR graph representation format; the end index is considered excluded from the partition. Edges of the pseudo-partition are contained in the slice of the column index vector between the pointers in the starting and ending positions of the row pointer vector. Figure 4.8 shows an example of how a pseudo-partition is extracted from a graph. We transfer both slices of nodes



(a) Pseudo-partition graph



(b) Pseudo-partition in CSR format

Figure 4.8. Pseudo-partition including nodes 2, 3 and 4.

and edges vectors to device memory, as well as the slices of the `colors` and `rand` vectors pertaining to the pseudo-partition. Because we can pseudo-partition the graphs with a node granularity, the algorithm fails the coloring if the device memory cannot contain $4(\Delta + 4)$ bytes, as the partition with a single node with the maximum degree cannot fit completely in memory.

Figure 4.9 reports the new definition of the kernel function for method D2, highlighting the differences with the original code in Figure 4.7. We take partitions into account by substituting the input parameter`n` with the two integer values `first` and `last` on line 2. The value of `n` is now the number of nodes in the pseudo-partition `last-first`, as shown on line 6. On lines 12 and 19 we restore pointers of the row pointer slice so that they correctly point to the column index slice. The resulting neighbor index `w` on line 19 is in the range $[0, last - first)$ if it is inside the pseudo-partition. On line 26 we ignore neighboring nodes that are outside of the pseudo-partition. Performances between kernels from Figures 4.7 and 4.9 are practically the same, as the only long operation we add is a single memory load on line 12. After all the pseudo-partitions are colored independently, we sequentially scan the `colors` vector in search of conflicting pairs of nodes. For each conflicting pair, we save the node with the lower index and erase its color. We then sequentially recolor the nodes that have been uncolored using the return value of the `suggest_vertex_color(int)`. This process of conflict search and resolution is performed only in case the graph has been partitioned; if the graph is entirely colored with no need for partitions, the conflict search is skipped. Since partitions are created in lexicographical order of the nodes, we expect a high number of conflicts. The overall time to perform coloring using this partitioning technique is high, but with it, we are able to color larger graphs.

4.6 Method E - Cohen-Castonguay (cuSPARSE)

The CuSparse class acts as a wrapper to the CSRCOLOR routine of the cuSPARSE library. The routine accepts graphs in CSR format, so the execution halts if a different format was chosen at compile time. We only override the `start_coloring()` method, to interface with the GPU, manage its memory, start the routine, and extract the result. We allocate and transfer to devise memory both the row pointers and column index vectors with CUDAMALLOC and CUDAMEMCPY. We also allocate memory for the `colors` vector to store the result, and for the vector of edge weights `weights`, which the routine requires but does not access. The vector of weights is of length m and contains a weight for each edge of the graph. Since it is required but not accessed, we decide to allocate memory, but we leave it uninitialized. Weighted graphs are outside the scope of this thesis, as weights do not influence the process of coloring. Other parameters to the routine include the percentage of nodes to color, which we set at 100%, and the output parameter $ncolors$ where the routine will store how many colors are used. Other cuSPARSE-specific

```
__global__ void JonesPlassmannLuby::color_kernel(
    int c, int first, int last, int* row_ptr,
    int* col_idx, int* rand, int* colors)
{
    for (int v = threadIdx.x + blockIdx.x * blockDim.x;
        i < last - first;
        i += blockDim.x * gridDim.x
    ) {
        int  color     = 2 * c;
        bool localmin  = true;
        bool localmax  = true;
        int idx_corrector = row_ptr[0];

        if (colors[v] != -1) continue; // Ignore colored nodes

        // Loop on neighbors
        for (int i = row_ptr[v]; i < row_ptr[v+1]; ++i) {
            // Valid neighbors are in range [0, last - first)
            int w  = col_idx[i - idx_corrector] - first;
            int cw = colors[w];

            int vr = rand[v];
            int wr = rand[w];

            // Ignore neighbors out of partition
            if (w < 0 || w >= last - first ||
                (cw != -1 && // Ignore colored neighbors
                // Consider neighbors colored this iteration
                cw != color && cw != color+1) ||
                v == w) // Ignore looping edges
                continue;

            // Determine inclusion in independent sets
            localmin &= vr < wr;
            localmax &= vr > wr;
        }

        // Assign color
        if (localmin) colors[v] = color + 1;
        if (localmax) colors[v] = color;
    }
}
```

Figure 4.9. Kernel function for method D2 with pseudo-partitioning.

parameters are needed and can be checked in the cuSPARSE user guide [2].

Calling the CSRCOLOR routine to color a graph requires $4\,(2n + 2m + 1)$ bytes

of memory, assuming the use of 32 bit integer and floating point numbers. On the other hand, using methods D1 and D2, we need $4\,(3n + m + 1)$ bytes in the device's global memory. By imposing the following inequality between the two

$$4\,(2n + 2m + 1) \geq 4\,(3n + m + 1)$$

and isolating n from m

$$2n - 3n \geq m - 2m$$

we conclude that method E requires more space in the device global memory that methods D when

$$n \leq m$$

; this is true for all graph we consider during the tests, listed in Table 5.1.

Analyzing the coloring produced by the routine, we notice that some colors lower than *ncolors* are not assigned to any vertex. We ascertain that *ncolors* is the maximum color and that the number of colors used is often smaller. We insert the values of the `colors` vector, after it has been retrieved from the device memory, into a set from the C++ standard library and take its size, which corresponds to the effective number of colors used.

4.7 Method F - Jones-Plassmann-Luby (Gunrock)

In this section, we present the implementation of the Jones-Plassmann-Luby algorithm present in the Gunrock library, despite it not being part of our software. *Gunrock* [21] is an open-source framework to ease the development of programs that perform complex computations on graphs over the GPU. The library is distributed with several example programs, which include the graph coloring implementation we consider. The implementation follows the best-performing algorithm from the research paper by Osama *et al.* [18]. We obtain and compile the code directly from the development branch of the Gunrock repository hosted on GitHub [22], whose last commit was dated 15th November 2021. The structure of the code for method F follows a design pattern imposed by the Gunrock framework, but the function executing an iteration of the algorithm—declared as a C++ lambda function—is very similar to the one presented in Figure 4.7 for method D2. The algorithm finds the two independent sets of maximum and minimum random values and assigns a color to each iteration. The sets are constructed by discarding nodes whose random number is non-maximum or non-minimum. Random values are generated on the GPU with the cuRAND library as uniformly distributed floating point numbers, stored in the `rand` vector. The function implementing the algorithm is dispatched with a *parallel for*—one of Gunrock's high-level API—on every node that has not been assigned a color. The *parallel for* issues a kernel launch on the GPU that dispatches a thread for every non-colored node, and executes the function implementing the JPL algorithm on every thread. A single thread performs work only

on a single node. The output of the coloring is reported in a JSON file and contains, among other data, the average number of colors and processing time between multiple runs of the algorithm that can be performed on the same execution.

While experimenting with the Gunrock coloring implementation, some executions would not terminate. The bug is an infinite loop caused by the `rand` vector, whose values are not unique. When two neighboring nodes are assigned an identical random value, both of them are discarded from both independent sets. This happens because the evaluation of a node's membership to a set is done by comparing two random values with strict $<$ and $>$ comparison operators. We then search the commit where the bug was introduced, knowing that the program had worked in the past, as it was the focus of the research paper by Osama *et al.* [18]. We find a commit dated 6th November 2019, where a lot of the code implementing graph coloring was refactored. Among the deleted lines of code, we find a peculiar instruction that enables a behavior not described in the research paper; the line of code performed a regeneration of the `rand` vector every two iterations. This would prevent two neighboring nodes with the same random value from never being colored, as the random values would change constantly, ultimately fixing the bug. We restore the lost behavior and submit a pull request to the repository that gets accepted soon after.

Chapter 5

Experimental Results

We run our experiments on a i9 10900KF CPU running at 3.7 GHz, with 10 cores and 20 threads, 64 GB of RAM, coupled with an NVIDIA RTX 3070 GPU with 5888 CUDA cores, and 8 GB of dedicated memory for embedding computations. We use Linux Ubuntu 20.04.3 LTS as the operating system. Experiments are performed on the set of graphs reported in Table 5.1. We include graphs from both real datasets (type **r**) and generated by an algorithm (type **g**). Some of the graphs are undirected (type **u**), while others are directed (type **d**); however, we consider the undirected counterpart of directed graphs, following the transformation described in Chapter 3.2. In this chapter, we vastly make use of our nomenclature to distinguish the many algorithms we consider, introduced in Table 4.1.

5.1 Experimental changes

In this section, we describe possible changes to be made in the implementations presented in Chapter 4.1. We present considerations backed by data harvested during the experimentation process.

5.1.1 Thread vs async

We experiment with method B3 from Table 4.1 by replacing threads with async objects. Async objects are similar to threads in the sense that allow parallel execution of a function, but return a future object used to retrieve return values or exceptions launched from the parallel function. Our experiment is carried out by launching a different number of either thread or async objects and analyzing the difference in runtime. We analyze 4 configurations using threads and 4 configurations using asyncs; the configurations vary in the number of threads or asyncs we launch in parallel. We use configurations threads or asyncs of 1, 2, 5 and 10 times the number of physical threads i.e. in our testing environment with 20 physical threads, configuration *thread5* launches 100 threads, and *async2* launches 40 asyncs. To account

Instance	n	m	$\overline{\delta}$	D	*Type*
af_shell3	504 855	17 588 875	34.8	6.90×10^{-5}	ru
apache2	715 176	4 817 870	6.7	9.42×10^{-6}	ru
ecology2	999 999	4 995 991	5.0	5.00×10^{-6}	ru
G3_circuit	1 585 478	7 660 826	4.8	3.05×10^{-6}	ru
offshore	259 789	4 242 673	16.3	6.29×10^{-5}	ru
parabolic_fem	525 825	3 148 801	6.0	1.14×10^{-5}	ru
thermal2	1 228 045	8 580 313	7.0	5.69×10^{-6}	ru
twitch_gamers [23]	168 114	13 595 114	80.9	4.81×10^{-4}	ru
ASIC_320ks	321 671	1 827 807	5.7	1.77×10^{-5}	rd
atmosmodd	1 270 432	8 814 880	6.9	5.46×10^{-6}	rd
cage13	445 315	7 479 343	16.8	3.77×10^{-5}	rd
email_Enron	36 692	367 662	10.0	2.73×10^{-4}	rd
FEM_3D_thermal2	147 900	3 489 300	23.6	1.60×10^{-4}	rd
thermomech_dK	204 316	2 846 228	13.9	6.82×10^{-5}	rd
rgg_n_2_15_s0	32 768	320 480	9.8	2.98×10^{-4}	gu
rgg_n_2_16_s0	65 536	684 254	10.4	1.59×10^{-4}	gu
rgg_n_2_17_s0	131 072	1 457 506	11.1	8.48×10^{-5}	gu
rgg_n_2_18_s0	262 144	3 094 566	11.8	4.50×10^{-5}	gu
rgg_n_2_19_s0	524 288	6 539 532	12.5	2.38×10^{-5}	gu
rgg_n_2_20_s0	1 048 576	13 783 240	13.1	1.25×10^{-6}	gu
rgg_n_2_21_s0	2 097 152	28 975 990	13.8	6.59×10^{-6}	gu
rgg_n_2_22_s0	4 194 301	60 718 396	14.5	3.45×10^{-6}	gu
rgg_n_2_23_s0	8 388 608	127 002 786	15.1	1.80×10^{-6}	gu
rgg_n_2_24_s0	16 777 216	265 114 400	15.8	9.42×10^{-7}	gu
qg.order100 [24]	10 000	1 980 000	198.0	1.98×10^{-2}	gd

Table 5.1. List of graphs used in the experiments. Unless otherwise stated, all graph instances are obtained through [25]

for fluctuations in runtime speeds, we run the algorithm with the same settings 10 times and present an average of the runtime results in Table 5.2 as the relative difference from the base configuration of *thread1*. In the table, negative numbers represent a slowdown and positive numbers represent a speedup with respect to the base configuration *thread1*. We realize that all configurations, on average, perform better than our baseline. The better configurations are the ones that use twice the number of physical threads as their number of threads or asyncs; other configurations oscillate with speedups and slowdowns across all instances of the graphs. Despite a significant slowdown of 35.96% and 24.79% for *thread2* and *async2* respectively on graph *twitch_gamers*, all other runs show positive speedups, with an average speedup of over 10% for both configurations. Despite the positive gains in

Instance	*thread1*	*asynch1*	*thread2*	*asynch2*	*thread5*	*asynch5*	*thread10*	*asynch10*
af_shell3	0.0%	−1.06%	10.42%	11.09%	18.79%	18.56%	20.57%	22.08%
apache2	0.0%	−1.65%	10.27%	15.83%	19.09%	13.74%	0.34%	6.52%
ecology2	0.0%	7.76%	19.27%	16.66%	18.93%	20.30%	21.13%	18.81%
G3_circuit	0.0%	−2.24%	2.14%	2.94%	−20.02%	−25.41%	−17.93%	−20.85%
offshore	0.0%	0.81%	10.42%	14.71%	17.08%	2.85%	9.99%	11.55%
parabolic_fem	0.0%	−1.35%	17.96%	23.11%	18.27%	21.39%	17.45%	16.04%
thermal2	0.0%	0.68%	10.82%	2.45%	−5.63%	−9.70%	8.21%	10.75%
twitch_gamers	0.0%	1.17%	−35.96%	−24.79%	−21.73%	−7.11%	−26.62%	−19.87%
ASIC_320ks	0.0%	1.19%	11.66%	4.36%	8.75%	12.91%	6.22%	7.74%
atmosmodd	0.0%	−0.68%	3.45%	10.95%	−9.31%	−19.60%	−5.69%	−1.96%
cage13	0.0%	0.60%	14.14%	19.35%	22.76%	20.87%	18.29%	13.19%
email_Enron	0.0%	−0.36%	8.32%	10.21%	11.50%	3.33%	−14.19%	−30.15%
FEM_3D_thermal2	0.0%	−1.52%	14.57%	13.73%	13.28%	16.18%	13.29%	17.99%
thermomech_dK	0.0%	0.46%	5.91%	11.78%	9.34%	7.37%	12.77%	6.06%
rgg_n_2_15_s0	0.0%	−2.67%	5.55%	6.56%	−0.26%	−5.63%	−32.23%	−36.05%
rgg_n_2_16_s0	0.0%	1.53%	5.98%	7.12%	−2.86%	−2.72%	−19.94%	−9.70%
rgg_n_2_17_s0	0.0%	−2.12%	13.42%	8.81%	14.49%	12.21%	7.06%	8.96%
rgg_n_2_18_s0	0.0%	−3.39%	14.41%	16.27%	9.54%	8.88%	7.87%	11.89%
rgg_n_2_19_s0	0.0%	−3.48%	14.33%	16.98%	19.09%	21.30%	20.41%	23.79%
rgg_n_2_20_s0	0.0%	2.01%	15.33%	13.34%	14.26%	13.28%	14.77%	16.99%
rgg_n_2_21_s0	0.0%	6.94%	20.36%	18.49%	19.77%	19.47%	16.35%	10.62%
rgg_n_2_22_s0	0.0%	−8.14%	8.11%	10.77%	−4.35%	−4.87%	−2.73%	−5.24%
rgg_n_2_23_s0	0.0%	6.02%	16.66%	17.79%	15.33%	5.90%	3.28%	11.09%
rgg_n_2_24_s0	0.0%	11.67%	19.74%	22.52%	6.16%	4.78%	6.59%	0.37%
qg.order100	0.0%	17.71%	30.18%	15.14%	38.12%	38.24%	16.74%	23.39%
average	0.0%	1.20%	10.70%	11.45%	9.22%	7.46%	4.08%	4.56%

Table 5.2. Relative percentage of runtime in different thread and async configurations. *thread1* is used as baseline.

speed shown by this experiment, we will continue to use the *thread1* configuration throughout this chapter, as the actual speedup in terms of absolute values is usually negligible, amounting to a few hundred milliseconds only for the larger *rgg* graphs.

5.1.2 Regeneration of `rand` vector

Methods D2 and F are respectively our implementation and Osama *et al.* [18]'s implementation with the Gunrock library of the Jones-Plassmann-Luby algorithm with two independent sets coloring. Comparing the iteration section of both codes, we notice that the Gunrock implementation adds an extra step at the beginning; every two iterations, the values in the `rand` vector are regenerated with the cuRAND library. We suspect this practice has the effect of creating larger independent sets, which in turn lowers the number of colors, as method D2 generates around 20% more colors than Method F. We want to experiment with this feature by adding it to our software, but we find it impossible to use the cuRAND library as it is done in method F. In fact, the `rand` vector of Gunrock/color contains random floating point values, which can be generated with uniform distribution with the cuRAND library; methods D1 and D2, instead, hold unique integers as a random permutation of vertex indexes. This core difference leads us to a different way of changing the contents of the `rand` vector. Our goal is to change the random number of every vertex. Each vertex is identified by a unique index v, and its random value is stored in the *rand* array at index v. By shifting the array by one position to the left, wrapping the first element to the last position, we achieve our goal of modifying all random numbers associated with the vertices. Let's call $rand\,[v]$ the value stored in the vth position of the *rand* array, and $rand_i$ the state of the array after i shifts to the left. We see that the random values associated with vertex v $\{rand_i\,[v] \mid i = 0 \dots n-1\}$ change with every shift, and repeat after n shifts have been performed. Shifting the vector in memory is a much simpler solution than completely regenerating it, but doing so in a GPU architecture, where many threads cooperate to solve tasks in parallel, still proves to be a difficult task. Instead of performing a shift, we notice that the same effect of shifting the array can be achieved by modifying the index. In fact, accessing position v after k shifts leads to the same value of accessing the kth index to the right of v, with proper wrapping to the start of the array after the end is reached; we can write it as $rand_k\,[v] = rand_0\,[v + k \bmod n]$. We call this technique **index shifting**, as opposed to the memory shifting technique we discussed earlier. We can implement index shifting by simply applying an additive constant when specifying which index to access. The constant would depend on the current number of iterations, and on the frequency of shifting. Following Gunrock/color's footsteps, we start by advancing the index by one position every two iterations. We also try other configurations, by changing the frequency of index shifting; we identify each configuration with the name of the method—either $D1$, $D2$, or F—with the subscript T indicating the

number of iterations that pass between one index shift and the next. In particular, $T = \infty$ refers to the standard configuration described in Chapter 4.5, where index shifting is not performed. Table 5.3 reports the number of colors we observe after running each configuration 10 times on the graphs in Table 5.1. From the table, we

Instance	$D1_\infty$	$D1_1$	$D1_2$	$D2_\infty$	$D2_1$	$D2_2$	F_2
af_shell3	66	59	65	66	46	49	49
apache2	19	18	19	19	12	13	16
ecology2	15	15	16	15	10	11	12
G3_circuit	15	15	16	15	10	11	11
offshore	37	24	26	37	23	23	27
parabolic_fem	18	12	13	18	12	12	13
thermal2	18	13	14	19	13	12	15
twitch_gamers	769	468	502	776	470	473	509
ASIC_320ks	22	18	19	22	15	15	18
atmosmodd	20	19	19	20	13	14	14
cage13	66	38	42	65	37	38	41
email_Enron	164	118	124	159	118	117	127
FEM_3D_thermal2	50	46	46	51	37	39	38
thermomech_dK	30	21	23	30	20	20	21
rgg_n_2_15_s0	24	21	21	24	20	20	20
rgg_n_2_16_s0	27	22	24	27	23	23	23
rgg_n_2_17_s0	29	24	24	29	24	24	26
rgg_n_2_18_s0	31	27	26	31	25	25	27
rgg_n_2_19_s0	32	27	28	32	26	27	29
rgg_n_2_20_s0	34	29	30	34	30	29	33
rgg_n_2_21_s0	36	30	30	36	30	30	32
rgg_n_2_22_s0	39	31	32	38	31	31	33
rgg_n_2_23_s0	40	33	34	40	33	33	34
rgg_n_2_24_s0	43	35	35	42	35	35	37
qg.order100	357	299	310	356	239	261	221

Table 5.3. Colors produced by configurations of index shifting

see how both $D1_\infty$ and $D2_\infty$ have the highest usage of colors among the considered configurations. We see that as we lower T, the number of colors gets smaller as well, but we cannot explain why this happens. In particular, configurations with $T = 1$ produce the least colors with both $D1$ and $D2$ methods for the majority of the test graphs; in the rare cases where this does not happen, the result is still plenty satisfying, as $T = 1$ uses only 1 more color than the corresponding $T = 2$ configuration. Overall, we find a reduction in the number of colors used of 19% from configuration $D1_\infty$ to $D1_1$, and a reduction of 28% from $D2_\infty$ to $D2_1$. The

reduction is more prominent with $D2$ rather than with $D1$ because, on average, method $D2$ creates independent sets larger than method $D1$. To explain this, let's consider two successive iterations of method $D1$ and a single iteration of method $D2$ that assign colors c and $c+1$ to the independent sets they find. For method $D2$, the two independent sets are disjoint, and created from the same subset of V. For method $D1$, the first independent set is created from a larger subset of V than the second independent set, which is bound to choose fewer nodes to add to the set. When the independent sets are smaller, more iterations need to be performed to color the whole graph, which increases the number of overall colors. Method $D2_1$ outperforms method F_2 by using 10% fewer colors on average. We know that Gunrock/color performs a regeneration of the `rand` vector every two iterations. Osama [26] explains that they see "minor performance improvement (in terms of the number of colors generated and elapsed time)", but do not share if they tried other configurations. We assume that they stop at one regeneration every two iterations to mitigate the effect of constantly writing data to global memory between each iteration. A positive side effect of index shifting is that no extra memory operations are needed, as we modify the index locally for every thread. This makes index shifting a far more appealing choice than regenerating the whole vector. We decide to keep using configurations $D1_1$ and $D2_1$ for methods D1 and D2 respectively, as they improve the quality of the coloring, with little to no repercussions in terms of runtime performance.

5.2 Results

In this section, we analyze the performance achieved by the methods listed in Table 4.1 on the graphs from Table 5.1, in terms of coloring runtime and quality of the solution, i.e. number of colors. The values we present are averaged across 10 runs of the method on the graph, to account for fluctuations in runtime speeds, as well as provide a better estimate of the number of colors used by non-deterministic algorithm implementations. In the following tables, we use τ_M and c_M to indicate the coloring time in ms and the number of colors respectively; M is the method the data refers to.

5.2.1 Results for CPU algorithms

Table 5.4 reports the performance of methods A1, A2, A3, and A4, implementing the sequential greedy algorithm on the CPU. We see that the times to color can be divided into two distinct groups: methods A1 and A2 are around 3.1 times faster than methods A3 and A4. The sort operation to generate the order in which nodes are to be colored, shown in Figure 4.4 is the cause of this behavior. For method A1, the call to the sort function could be avoided, as nodes are already in the desired order. For method A2, the desired order is the reverse of the starting order; we

Instance	τ_{A1}	c_{A1}	τ_{A2}	c_{A2}	τ_{A3}	c_{A3}	τ_{A4}	c_{A4}
af_shell3	477.65	25	462.60	25	1068.5	29	1069.9	33
apache2	321.87	3	283.50	3	1184.7	7	1180.7	7
ecology2	422.48	2	382.43	2	1625.5	5	1589.5	5
G3_circuit	686.89	4	618.61	3	2659.1	5	2747.2	6
offshore	167.15	12	148.10	12	451.98	11	452.00	14
parabolic_fem	242.68	5	219.13	6	829.36	7	821.95	7
thermal2	602.86	7	539.39	7	2115.8	7	2102.7	7
twitch_gamers	293.07	117	292.02	124	515.59	116	504.71	135
ASIC_320ks	140.89	8	131.06	6	482.68	6	475.87	8
atmosmodd	588.65	2	533.93	2	2244.7	7	2251.7	7
cage13	275.61	16	251.96	16	823.70	15	783.78	18
email_Enron	17.466	35	16.324	54	50.293	29	51.176	54
FEM_3D_thermal2	104.45	8	101.70	8	259.61	18	257.79	18
thermomech_dK	122.21	14	115.59	14	331.54	14	331.49	14
rgg_n_2_15_s0	16.245	13	15.011	14	47.062	14	47.154	17
rgg_n_2_16_s0	33.679	15	30.974	16	99.915	14	100.75	18
rgg_n_2_17_s0	70.032	16	66.327	15	217.50	15	216.98	19
rgg_n_2_18_s0	144.91	17	136.77	17	447.25	16	444.64	21
rgg_n_2_19_s0	300.00	19	282.51	19	970.51	18	1000.4	22
rgg_n_2_20_s0	620.54	18	563.40	20	2110.3	18	2050.6	25
rgg_n_2_21_s0	1282.3	21	1204.7	19	4360.5	19	4381.9	23
rgg_n_2_22_s0	2557.0	21	2471.4	22	9081.5	20	9180.2	25
rgg_n_2_23_s0	5472.3	23	4931.1	22	19461	21	19767	27
rgg_n_2_24_s0	11331	23	10531	23	41605	22	41608	27
qg.order100	37.318	128	372.14	128	44.741	125	44.818	125

Table 5.4. Coloring times and number of colors for Greedy algorithm

could substitute the sort function with the faster `reverse` function from the C++ standard library. With both methods, we prefer to maintain the call to the sort function, so that all four different methods share some part of the codebase for higher readability. This also allows us to possibly change the way we initialize the `vert_copy` vector, without affecting the correctness of the algorithm. On the other hand, with methods A3 and A4, calling the sort function performs a non-trivial sort operation, which increases the necessary time. Regarding the number of colors used, method A1 produces the best solution among the most graph instances. Method A1 is beaten by method A3 on graphs `offshore`, `twitch_gamers`, `ASIC_320ks`, `cage13`, `email_Enron`, `qg.order100` and all *rgg* graphs other than `rgg_n_2_15_s0` and `rgg_n_2_20_s0`. The margin in colors, however, is not very large, reaching a maximum of 6 fewer colors for graph `email_Enron`. We find mixed results for

method A2, which colors some graphs as good as the best solutions from method A1—or even better in the case of graph `G3_circuit`—and other graphs present a number of colors comparable to or worse than a bad solution from method A3. Method A4 presents terrible results in terms of the number of colors, producing the worst solutions for all graphs other than `qg.order100`, where the solution produced is tied for the best with method A3. Overall, our results do not completely match previous research [12], stating that a First Fit rule (method A1) usually produces worst results than a Largest Degree First ordering (method A3); this is possibly caused by either our chosen set of test graphs or by a different policy in ordering nodes with the same degree in method A3.

Instance	τ_{B1}	c_{B1}	τ_{B2}	c_{B2}	τ_{B3}	c_{B3}	τ_{B4}	c_{B4}
af_shell3	1468.1	27	1387.7	26	63.393	27	237.91	24
apache2	1821.5	6	1888.6	5	30.104	4	284.93	5
ecology2	2567.9	5	2679.5	5	36.729	4	405.29	4
G3_circuit	4131.8	6	4267.0	5	58.879	5	628.34	5
offshore	669.43	12	695.62	11	18.695	13	114.75	11
parabolic_fem	1366.7	6	1371.0	6	22.761	6	214.22	6
thermal2	3197.6	8	3285.3	7	59.966	7	500.95	6
twitch_gamers	480.67	117	525.55	112	51.394	118	114.19	115
ASIC_320ks	836.06	9	833.46	6	13.288	8	134.06	6
atmosmodd	3295.9	6	3434.6	5	58.627	6	530.12	5
cage13	1133.5	15	1182.9	14	32.676	16	191.86	14
email_Enron	92.818	37	99.690	36	3.1574	42	16.970	44
FEM_3D_thermal2	382.86	16	396.37	15	13.583	18	67.361	15
thermomech_dK	523.97	14	540.76	12	13.359	13	88.951	12
rgg_n_2_15_s0	83.227	14	87.677	13	2.1415	14	13.946	13
rgg_n_2_16_s0	169.77	15	174.27	15	3.9013	17	28.448	15
rgg_n_2_17_s0	340.59	16	342.79	15	7.9251	16	54.945	15
rgg_n_2_18_s0	662.25	17	684.74	17	15.601	19	113.08	17
rgg_n_2_19_s0	1357.0	18	1397.5	18	31.939	19	221.82	18
rgg_n_2_20_s0	2713.2	19	2806.8	18	66.060	19	451.10	19
rgg_n_2_21_s0	5554.3	20	5814.4	19	139.64	20	955.57	19
rgg_n_2_22_s0	11352	21	11988	20	285.03	22	1851.8	20
rgg_n_2_23_s0	22991	23	24789	22	592.69	24	3807.4	22
rgg_n_2_24_s0	48026	23	52785	23	1221.1	23	7748.0	22
qg.order100	75.717	140	51.979	121	15.649	147	14.683	117

Table 5.5. Coloring times and number of colors for Gebremedhin-Manne algorithm

Table 5.5 reports the performance of methods B1, B2, B3, and B4, implementing the parallel Gebremedhin-Manne algorithm on the CPU. We see that the two

synchronous versions of the algorithm, methods B1 and B2, are largely slower than the asynchronous versions of methods B3 and B4. The original paper [12] mentions that "the asynchronous version runs faster by a factor of 3 to 5"; our results show a speedup factor of 4 to 7 for the improved versions of the algorithm, methods B2 and B4. Instead, the standard version B3 presents a much higher speedup with respect to the synchronous standard version of method B1, going from a minimum of 4.8 times for graph `qg.order100` to 70.1 times for graph `G3_circuit`, averaging a speedup of 41.9 times faster. The original paper also mentions that the time needed for synchronization with the barrier increases with the number of threads performing the computation. In their experiments, Gebremedhin and Manne vary the number of processors used to a maximum of 12, reporting a speedup the more processors are used; instead, we use 20 threads, which may be too high of a number to benefit from the parallel execution, and possibly lose time due to the more prominent barrier synchronization in the two synchronous methods B1 and B2. Regarding the number of colors, we confirm the initial assumption that the improved algorithm decreases or leaves unchanged the number of colors generated by the first step of the algorithm. It must be noted, however, that this does not fully translate to the improved algorithm generating fewer colors than the standard algorithm. Method B4 colors graphs `apache2` and `email_Enron` with respectively 1 and 2 more colors than B3, but this is an isolated case. For all other combinations, the improved algorithm does produce a number of colors less or the same as the standard algorithm. The cost of lowering the number of colors is to increase the overall runtime. Comparing the best solution for methods A and B—A1 and B3—we report that method B3 is around 8.8 times faster, probably because it uses multiple threads, but produces 24% more colors on average.

Table 5.6 reports the performance of methods C1 and C2, implementing the parallel Jones-Plassmann algorithm on the CPU. Data shows slight to no difference when comparing the runtime of the two methods for the majority of graph instances. Method C2 gains a noticeable speedup on method C1 on those graphs whose nodes present a wide range of degrees; namely, *cage13* is colored 11.5% faster, `twitch_gamers` is colored 32.3% faster, and `email_Enron` is colored 49.7% faster. This is caused by the different ways we divide nodes into partitions. Method C1 creates partitions with roughly the same number of nodes, while method C2 creates partitions containing roughly the same number of edges. For graphs where edges are distributed more or less equally across nodes, the two methods create similar partitions. On the other hand, graphs whose nodes present a wide range of degrees, where edges are concentrated on the earlier nodes in lexicographical order, are divided into highly unbalanced partitions by method C1. Given that the work performed by each thread is linear in the number of edges of its partition, having partitions with a similar number of edges helps in having all threads terminate in a short window of time. If partitions have different amounts of edges, however, threads working on partitions with fewer edges will terminate earlier than threads

Instance	τ_{C1}	c_{C1}	τ_{C2}	c_{C2}
af_shell3	93.692	28	94.373	27
apache2	41.902	7	41.881	7
ecology2	49.075	5	49.455	5
G3_circuit	78.084	6	83.631	6
offshore	27.805	13	27.846	13
parabolic_fem	31.667	7	31.616	7
thermal2	74.601	7	75.436	7
twitch_gamers	250.08	121	169.30	119
ASIC_320ks	17.548	8	17.498	8
atmosmodd	75.162	7	74.906	6
cage13	58.571	16	51.829	16
email_Enron	12.841	42	6.4488	40
FEM_3D_thermal2	20.287	17	20.119	18
thermomech_dK	19.763	13	19.726	12
rgg_n_2_15_s0	2.7021	15	2.6511	15
rgg_n_2_16_s0	5.5171	18	5.2675	18
rgg_n_2_17_s0	10.816	17	10.732	17
rgg_n_2_18_s0	22.123	19	22.099	19
rgg_n_2_19_s0	47.346	19	46.972	19
rgg_n_2_20_s0	97.171	20	97.523	21
rgg_n_2_21_s0	201.45	20	201.30	22
rgg_n_2_22_s0	425.44	22	427.41	22
rgg_n_2_23_s0	900.59	23	893.36	23
rgg_n_2_24_s0	1877.9	25	1878.8	24
qg.order100	12.447	109	12.424	107

Table 5.6. Coloring times and number of colors for Jones-Plassmann algorithm

working on partitions with more edges, thus wasting computational resources. As an example to prove this conjecture, we provide that over 50% of nodes on the `twitch_gamers` graph is contained in the first 25% of nodes, in lexicographical order; this is why we notice a substantial speedup. The number of colors generated by both methods on each graph is more or less the same. We consider method C2 to be the best implementation between the two, as it is better on a wider number of graphs. Overall, method C2 is faster than the sequential implementation of method A1 but slower than the parallel implementation of method B3. The number of colors is always not better than methods A1 and B3, other than for the `qg.order100` graph, colored by method C2 with 14% and 8% less colors than methods A1 and B3 respectively.

5.2.2 Results for GPU algorithms

Instance	τ_{D1}	c_{D1}	τ_{D2}	c_{D2}	τ_E	c_E	τ_F	c_F
af_shell3	12.576	60	5.8255	46	5.9724	80	28.085	49
apache2	2.0726	18	0.8742	12	3.8861	33	1.1866	16
ecology2	1.8826	15	0.7561	10	3.2178	32	0.8714	12
G3_circuit	2.2633	15	0.9606	10	4.1577	32	1.1841	11
offshore	1.9085	24	1.0259	24	3.4819	48	3.2902	27
parabolic_fem	1.5309	12	0.8694	12	2.7860	32	1.0553	13
thermal2	2.3871	12	1.3154	12	4.5195	33	2.0279	15
twitch_gamers	2588.6	476	1224.6	469	75.595	504	2774.6	509
ASIC_320ks	3.5919	17	1.7818	15	3.7506	48	3.4160	18
atmosmodd	2.6420	18	1.1800	13	4.5680	35	1.7150	14
cage13	3.1115	37	1.7160	37	4.7012	64	5.7199	41
email_Enron	26.931	119	13.284	117	9.5010	146	36.738	127
FEM_3D_thermal2	2.2242	44	1.1346	36	4.2934	64	3.4798	38
thermomech_dK	1.2694	21	0.7175	20	3.4366	48	2.0432	21
rgg_n_2_15_s0	0.4431	21	0.2523	21	3.1055	48	0.6031	20
rgg_n_2_16_s0	0.5602	23	0.3206	23	3.2023	48	0.7585	23
rgg_n_2_17_s0	0.8443	24	0.4841	24	3.2511	48	1.1635	26
rgg_n_2_18_s0	1.3747	25	0.7774	25	4.4512	50	2.1031	27
rgg_n_2_19_s0	3.6363	27	1.9406	27	3.9091	48	4.4323	29
rgg_n_2_20_s0	5.9281	29	3.1602	29	5.7149	57	8.3825	33
rgg_n_2_21_s0	11.079	30	5.7060	30	8.6801	59	16.899	32
rgg_n_2_22_s0	22.753	31	11.634	31	15.287	64	35.851	33
rgg_n_2_23_s0	50.776	33	25.713	33	38.113	64	77.216	34
rgg_n_2_24_s0	115.53	35	57.995	35	75.728	64	167.91	37
qg.order100	25.428	311	10.378	137	12.988	129	35.514	221

Table 5.7. Coloring times and number of colors for coloring algorithms running on the GPU.

Table 5.7 reports the performance of methods D1, D2, and F, implementing the parallel Jones-Plassmann-Luby algorithm, and method E, implementing the parallel Cohen-Castonguay algorithm, on the GPU. We see that method D2 completes the coloring between 1.7 and 2.4 times faster than method D1, for an average of 1.99 times faster. This aligns with our expectations presented earlier in Chapter 4.5 because method D2 selects and colors two independent sets at the same time with a constant time difference from method D1, which only colors one. Method E is the fastest GPU algorithm on `twitch_gamers` and `email_Enron` graphs, but method D2 is on average 3.8 times faster than method E on the other graphs. As we use the CSRCOLOR routine from the cuSPARSE library as a black box for method E, we

do not know what causes the speedup on the two mentioned graphs. Method F is always slower than our implementation of the same algorithm in method D2, which proves to be 2.5 times faster on average. Interestingly, method D1 is 1.5 times faster than method F on all graphs other than `apache2`, `ecology2`, `G3_circuit`, `parabolic_fem`, `thermal2`, `ASCI_320ks` and `atmosmodd`, despite performing coloring in a less efficient manner. We already discussed differences in colors produced by methods D1, D2, and F in Chapter 5.1.2. Method D2 uses on average 10% fewer colors than both D1 and F; on larger graphs such as `twitch_gamers`, `cage13`, `email_Enron` and the *rgg* set, method D1 produces on average 7% fewer colors than method F. Method E always produces solutions that contain 118% more colors than method D2, except for graph `qg.order100`, where it uses 6% fewer colors. Method E likely produces solutions with this many colors because the Cohen-Castonguay algorithm implemented assigns 16 colors per iteration—as we proved in Chapter 3.7—to reduce the number of iterations performed, consequently reducing the coloring time. Comparing the CPU algorithms to method D2, we see how it is on average over 200 times faster than method A1, around 22 times faster than method B3, and 30 times faster than method C2. This result is to be expected, as GPU architectures provide a larger grade of parallelism compared to multi-core CPUs. What is not expected, however, is that CPU-based graph coloring is faster than our method D2 in graphs `twitch_gamers` and `email_Enron`: method A1 is 4.2 times faster, method C2 is 7.2 times faster and method B3 is 24 times faster than method D2 to color the first graph. The second graph is colored 4.2 times faster by method B3 and 2 times faster by method C2, while method A1 is slower than method D2. This surprising result is probably caused by the structure of the graphs; both graphs present a wide excursion of degrees between their nodes. Graph `twitch_gamers` has a maximum and minimum degree of $\Delta_t = 35279$, $\delta_t = 1$, and graph `email_Enron` has a maximum and minimum degree of $\Delta_e = 1383$, $\delta_e = 0$. This high discrepancy in the number of edges the GPU algorithm has to consider increases the effect of divergence, as well as requires many load instructions from the global memory, whose results cannot be cached in the limited cache memories of GPUs. The number of colors produced by method D2 on the GPU is between 40% and 50% greater than methods A1, B3, and C2 over all the graphs. Method B3, however, uses 7% more colors than method D2 on graph `qg.order100`, because it implements the standard Gebremedhin-Manne algorithm, which does not refine the coloring.

To investigate and motivate the speedup method D2 registers over method F, we profile both executables with the Nsight Systems profiler distributed by NVIDIA. We compare the coloring graph `af_shell3`, as it presents a prominent speedup, but it is colored roughly with the same amount of colors, hence in the same number of iterations. The profiler works by executing each kernel multiple times with the same input data, to collect all information needed without interfering too much with the timing of a single kernel call. We profile a single run of both algorithms; this

profiled run is different from the executions reported in Table 5.7, so results do not coincide perfectly. The profiler shows us that many kernels contribute to providing a coloring solution, i.e. the `count` function from the Thrust library launches two different kernels to return the number of non-colored nodes in method D2. On the profile for method F, we recognize the main kernel that computes the iteration, called `Kernel`, and another kernel called in between once every two `Kernel` calls, called `gen_sequenced`; this is the kernel that performs the regeneration of the array of random values. We also see other minor kernels, but we do not know what their purpose is. We only consider kernels that actively perform coloring—`color_kernel` for method D2 and `Kernel` for method F—as all other ones have a fast and fairly constant execution time. In Figure 5.1 we plot the execution times of each coloring

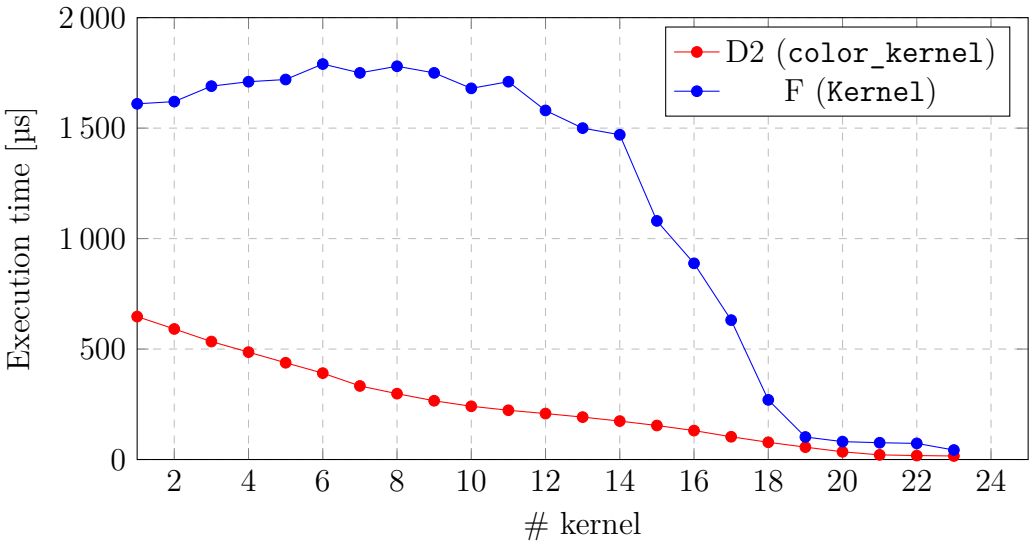


Figure 5.1. Execution times to complete each kernel

kernel in µs for both methods D2 and F. It is evident from the chart that method F's bottleneck is in the kernel function. Execution time for method D2's `color_kernel` is a slowly decreasing curve with respect to the iteration number, from 647 µs to 16 µs. On the other hand, the kernel function from method F executes roughly in constant time for the first 14 calls, at around 1669 µs, then the execution time quickly drops, stabilizing it the last 5 iterations around 75 µs. We find the cause of this not-so-great performance in two characteristics of the code provided by Gunrock. First, we notice from the profiler that, as the coloring progresses, the kernel function spawns a grid with fewer and fewer threads. This causes fewer warps to be ready to start executing when another warp is scheduled out of a SM while waiting for a global memory load. Graph algorithms rely heavily on memory operations, which can become a bottleneck if not properly balanced with computing

operations on GPU hardware. The second cause is in the formulation of the kernel function itself. In the code fragment that decides whether the current node is part of either independent set, access to the `rand` vector is performed 4 times per neighboring node, as no number is saved as a local variable in register memory. This is a huge problem for memory access, especially not saving the random value of the current node that is being colored, as it is constant between the whole execution of the kernel and does not need to be reloaded. Better memory management allows to access global memory once per neighbor, plus one time to load the current node v's random value, for a total of $\delta(v) + 1$ global memory accesses, as long as the loaded values are saved in register memory. The combination of these two behaviors leads the kernel to run out of ready warps while loading the random values, thus stalling the process. Between iterations 14 and 18, when the majority of nodes are colored, the balance that keeps the kernel stalled is broken, and the kernel can operate at a time comparable to the time of method D2. Regarding method D2, we designed the kernel to be launched on same-sized grids. Our implementation requires $2\delta(v)$ global memory loads per iteration on lines 22 and 23 of Algorithm 4.9, which is not optimal, but in our opinion improves code readability by having the two variables defined near each other.

Chapter 6

Conclusion

In this thesis, we analyze existing algorithms to solve the problem of graph coloring and propose, for the Jones-Plassmann-Luby algorithm we implement in CUDA, the technique of index shifting. By shifting the index by which threads access random numbers, we are able to reduce the number of colors used by around 28%, with a consequent increase in execution speed. Our implementation of the Jones-Plassmann-Luby algorithm with local minimum and local maximum independent sets reaches execution speeds of 2.5 times faster than previous state-of-the-art implementation based on the Gunrock framework; it also produces 10% fewer colors. However, different algorithms, such as the Gebremedhin-Manne algorithm we implemented for CPU architectures, produce solutions with 87% fewer colors, despite needing more time to complete. Further studies could implement more algorithms for GPU processors, allowing for a more fair comparison between the algorithms than the one we present. We conclude that moving highly parallel portions of algorithms to the GPU, a trend that is being explored in recent years, is a push in the right direction, as the benefits of having a faster code are immediately noticeable; on the other hand, however, GPU programming is a field that requires deep knowledge from the developer's part, and is in no way a simple feat to conquer. We hope that further advancements in technology can improve results obtained with GPU programming, while also decreasing the effort required from the developer.

Bibliography

[1] Wen-mei W. Hwu David B. Kirk. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann Publishers, 2010.

[2] NVIDIA Corporation. *CUDA Toolkit Documentation*. URL: `https://docs.nvidia.com/cuda/index.html` (visited on 04/30/2022).

[3] Khronos Group. *Khronos OpenCL Registry*. `https://registry.khronos.org/OpenCL/`.

[4] Frank Thomson Leighton. «A graph coloring algorithm for large scheduling problems». In: *Journal of research of the national bureau of standards* 84.6 (1979), p. 489.

[5] Gregory J Chaitin et al. «Register allocation via coloring». In: *Computer languages* 6.1 (1981), pp. 47–57.

[6] Fusun Akman. «Partial chromatic polynomials and diagonally distinct Sudoku squares». In: *arXiv preprint arXiv:0804.0284* (2008).

[7] Thomas F Coleman and Jorge J Moré. «Estimation of sparse Hessian matrices and graph coloring problems». In: *Mathematical programming* 28.3 (1984), pp. 243–270.

[8] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. «What color is your Jacobian? Graph coloring for computing derivatives». In: *SIAM review* 47.4 (2005), pp. 629–705.

[9] M. Garey and D. Johnson. *Computers and Intractability – A Guide to the Theory of NP-completeness*. Freeman, 1979.

[10] Dominic JA Welsh and Martin B Powell. «An upper bound for the chromatic number of a graph and its application to timetabling problems». In: *The Computer Journal* 10.1 (1967), pp. 85–86.

[11] Daniel Brélaz. «New methods to color the vertices of a graph». In: *Communications of the ACM* 22.4 (1979), pp. 251–256.

[12] Assefaw Hadish Gebremedhin and Fredrik Manne. «Scalable parallel graph coloring algorithms». In: *Concurrency: Practice and Experience* 12.12 (2000), pp. 1131–1146.

[13] Michael Luby. «A simple parallel algorithm for the maximal independent set problem». In: *SIAM journal on computing* 15.4 (1986), pp. 1036–1053.

[14] Mark T. Jones and Paul E. Plassmann. «A Parallel Graph Coloring Heuristic». In: *SIAM Journal on Scientific Computing* 14.3 (1993), pp. 654–669. DOI: 10.1137/0914041. eprint: https://doi.org/10.1137/0914041. URL: https://doi.org/10.1137/0914041.

[15] Jonathan Cohen. «Proof of optimality of minmax PIS algorithm». 2011. URL: https://jcohen.name/papers/Cohen_minmax_2011.pdf.

[16] Jonathan Cohen and Patrice Castonguay. «Efficient graph matching and coloring on the gpu». In: *GPU Technology Conference*. 2012, pp. 1–10.

[17] Maxim Naumov, Patrice Castonguay, and Jonathan Cohen. «Parallel graph coloring with applications to the incomplete-lu factorization on the gpu». In: *Nvidia White Paper* (2015).

[18] Muhammad Osama et al. «Graph coloring on the GPU». In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2019, pp. 231–240.

[19] Borione Alessandro. *GConv*. URL: https://github.com/LeaX-XIV/gconv (visited on 07/12/2022).

[20] George Karypis et al. «Multilevel hypergraph partitioning: Application in VLSI domain». In: *Proceedings of the 34th annual Design Automation Conference*. 1997, pp. 526–529.

[21] Yangzihao Wang et al. «Gunrock: GPU graph analytics». In: *ACM Transactions on Parallel Computing (TOPC)* 4.1 (2017), pp. 1–49.

[22] Muhammad Osama, Serban D. Porumbescu, and John D. Owens. «Essentials of Parallel Graph Analytics». In: *Proceedings of the Workshop on Graphs, Architectures, Programming, and Learning*. GrAPL 2022. May 2022, pp. 314–317. DOI: 10.1109/IPDPSW55747.2022.00061. URL: https://escholarship.org/uc/item/2p19z28q.

[23] Benedek Rozemberczki and Rik Sarkar. *Twitch Gamers: a Dataset for Evaluating Proximity Preserving and Structural Role-based Node Embeddings*. 2021. arXiv: 2101.03091 [cs.SI].

[24] Carla Gomes. *COLOR02/03/04: Graph Coloring and its Generalizations*. URL: https://mat.gsia.cmu.edu/COLOR02/ (visited on 04/07/2022).

[25] Timothy A. Davis and Yifan Hu. «The University of Florida Sparse Matrix Collection». In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011). ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. URL: https://doi.org/10.1145/2049662.2049663.

[26] Muhammad Osama. *Private Communications*. June 2022.