

本科生毕业设计

|  |
| --- |
| 基于并行计算架构的大规模图着色算法 |

|  |  |
| --- | --- |
| 院 系 | 计算机科学与技术 |
| 专业班级 | 卓越1501 |
| 姓 名 | 魏硕 |
| 学 号 | U201514482 |
| 指导教师 | 石宣化 |

2019年06月10日

**学位论文原创性声明**

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名： 年 月 日

**学位论文版权使用授权书**

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保密囗，在 年解密后适用本授权书

2、不保密囗 。

（请在以上相应方框内打“√”）

作者签名： 年 月 日

导师签名： 年 月 日

摘 要

随着当前信息时代的日益发展，数据量的逐渐增大，大规模图数据的处理在网页搜索应用，社交网络和交通网络等场景中作用也愈发明显。因此，大规模图着色的研究也开始成为高性能计算领域的热点。而通用GPU（Graphics Processing Unit）的发展使其作用逐渐广泛，用于图着色的加速也成了一种研究的趋势。但调研后，现有的较广泛的并行图着色均存在以下问题中的部分或全部：反复着色，在着色时，后续迭代轮次中顶点颜色会导致已着色顶点颜色出错，需要重新修改；并行度不够，每轮着色中，正确的着色顶点比例稀少；长尾问题，在着色最后，每轮着色顶点数大幅度减少；负载不均衡，每个线程运行时间开销相差过大，导致整体性能较差。

优化的新的基于并行架构的图着色算法，解决了反复着色，并行度不够以及长尾的问题。首先，预处理图数据，将其变成只有小编号顶点指向大编号顶点的有向图，避免了反复着色，也确保了算法能在有限轮次后终止。此外，配合使用CSR和CSC的存储格式保存图数据，可以做到一定程度的连续访存，并减少修正工作一半的工作量。其次，着色过程采用无冲突的并行着色思路，每一轮分为着色和修正两个步骤，提高了并行度，并可以避免原子操作。最后，在未着色顶点数剩余极少时，避免多次迭代，将着色策略变成串行，解决长尾问题。将其与一种典型的以边为中心的的算法融合互补，形成的组合算法解决了各自的主要缺点。

本算法使用CUDA C语言编程设计完成，提供了针对不同的图数据的接口参数。通过在相同环境下针对普遍的大规模图数据进行的实验，结果表明，该算法相较于CPU下的贪心着色算法有1.51-125.81倍的性能提升，相较于JPL（Jones-Plassman-Luby）有8.37-71.07倍的性能提升。

**关键词：**图着色；图形处理器；并行运算技术；两阶段

Abstract

With the development of the current information age, the amount of data has increased gradually, and the processing of large-scale map data has become more obvious in the fields of web search applications, social networks and transportation networks. Therefore, the study of large-scale graph coloring has also become a hot spot in the field of high-performance computing. The development of the general-purpose GPU (Graphics Processing Unit) has become more widespread, and the acceleration for graph coloring has become a research trend. However, after the investigation, the existing extensive parallel graph coloring has some of the following problems: repeated coloring, when coloring, the color of the vertex in the subsequent iteration round will cause the color of the shaded vertex to be wrong and needs to be re-modified; In each round of coloring, the correct coloring vertex ratio is rare; at the end of coloring, the number of colored vertex per round is greatly reduced; the load is unbalanced.

Optimized new graph rendering algorithm based on parallel architecture solves the problem of repeated coloring, insufficient parallelism and long tail. First, the map data is preprocessed, turning it into a directed graph with only small numbered vertices pointing to the large numbered vertices. In addition, with the storage format of the CSR and CSC storage format data, a certain degree of continuous memory access can be achieved, and half the workload of the correction work can be reduced. Secondly, the coloring process adopts a collision-free parallel coloring idea. Each round is divided into two steps of coloring and correction, which improves the degree of parallelism and avoids atomic operations. Finally, when there are few remaining uncolored vertices, avoid multiple iterations, turn the coloring strategy into serial, and solve the long tail problem. Combining it with a typical edge-centered algorithm complements the resulting combination of algorithms to solve their main shortcomings.

The algorithm is designed using CUDA C language programming and provides interface parameters for different data. Experiments on general large-scale graph data in the same environment show that the algorithm has 1.51-125.81 times performance improvement compared with the greedy coloring algorithm under CPU, compared to JPL (Jones-Plassman-Luby). There is a performance improvement of 8.37-71.07 times.

**Keywords：**Graph Coloring， GPU，CUDA，Two stages

目 录

[摘 要 I](#_Toc10369180)

[Abstract II](#_Toc10369181)

[1 绪 论 1](#_Toc10369182)

[1.1 课题背景 1](#_Toc10369183)

[1.2 国内外研究现状 1](#_Toc10369184)

[1.3 研究目的和主要内容 4](#_Toc10369185)

[1.4 论文结构 6](#_Toc10369186)

[2 相关背景 7](#_Toc10369187)

[2.1 CUDA相关知识 7](#_Toc10369188)

[2.2 数据存储 13](#_Toc10369189)

[2.3 难点分析 14](#_Toc10369190)

[2.4 本章小结 16](#_Toc10369191)

[3 算法实现 17](#_Toc10369192)

[3.1 算法思路 17](#_Toc10369193)

[3.2 算法分析 22](#_Toc10369194)

[3.3 算法关键技术 24](#_Toc10369195)

[3.4 本章小结 29](#_Toc10369196)

[4 性能评估 30](#_Toc10369197)

[4.1 实验设置 30](#_Toc10369198)

[4.2 切换执行阶段的时间 31](#_Toc10369199)

[4.3 与最新的技术比较 34](#_Toc10369200)

[4.4 在不同的GPU设备上运行本算法 35](#_Toc10369201)

[4.5 本章小结 36](#_Toc10369202)

[5 总结与展望 37](#_Toc10369203)

[致 谢 39](#_Toc10369204)

[参考文献 40](#_Toc10369205)

# 绪 论

本章我们首先介绍了当前发展图着色算法的目的以及意义，介绍了国内外在图着色领域的相关研究工作，并对算法的主要研究内容及工作意义作了具体说明，最后简略介绍了论文的整体结构。

## 课题背景

当前，随着互联网的大规模普及，社会面向数字化的变迁以及经济的迅猛发展，表达数据之间关联性的图数据的规模正在呈指数级增长。由于图计算能够用于分析数据之间的关联关系，因此其在互联网应用、科学计算、社会计算、商业计算等诸多领域得到应用广泛，已经成为大数据处理的主流模式之一。而图的着色算法便是其中的一种，其在时间规划调度，频道分配问题安全装箱问题等实际问题方面有着极为重要的作用，其中与图着色联系最紧密，实际应用最为广泛的即为寄存器分配技术。比如，相交图（interference graph）是用来表示程序变量是否拥有相交的生命期的。而一个寄存器同时只能被一个变量使用，如果用颜色表示寄存器，用顶点表示变量，该问题可抽象成一个图着色问题。

研究表明，在众多的图着色算法中，启发式算法和基于贪婪法的算法能够获得较好的着色数。然而，基于贪婪法的图着色算法具有O(n^2)的时间复杂度。由于实际图的大小不断增长，即使是具有线性时间的算法也需要求助于并行计算来缩减实际的求解时间。

本课题旨在重新审视并行计算环境中着色的问题，，拟设计并实现一种基于并行计算架构的近似最优解的图着色算法，帮助程序开发人员提高应用程序的并行性能。特别是，由于现实生活中的图遵循幂律属性，可以在各种图应用中发现该特点，如社交网络和生物医学网络分析，以及计算语言学，本文着重于加速幂律图上的图着色。

## 国内外研究现状

图着色问题是一个NP问题，即寻找一种着色方案使得图中相邻的顶点着色不同，在保证正确性的条件尽可能减少着色数目。在非并行情况下，经典的算法为按照顶点编号对顶点进行依次着色，对于无法进行下去的情况进行一定的回溯，能够进行优化就是进行一定的剪枝。

近年来，关于图着色问题，有多种算法被提出，它们对并行架构下的图着色的研究大多数主要集中在算法设计以及如何得到着色数[7,10]。

FastColor是一种基于迭代的适用于非常大的图的图着色算法[11]。FastColor通过计算工作图Gk的集合来设置下限，工作图Gk是从给定图G减小得到的，并且在算法执行期间逐步变小（最小独立集）。FastColor通过着色图Gm设置上限，Gm是收集所有从给定图G中移除的顶点和边得到的。通过迭代着色Gk和Gm，FastColor可以在上限与下限匹配时获得最终颜色计划。实验结果表明，FastColor可以在几分钟内为千万个顶点和一亿个边的图着色。

着眼于动态图着色问题，[14]中的工作提出了一种基于颜色传播的算法，该算法根据顶点的传播顺序对顶点进行着色。作者使用有向无环图作为给定图的辅助图。作者通过探索活动顶点的2跳邻居中的所有顶点，首先为辅助图着色。在基于颜色传播的算法中，仅当邻居已经改变时，顶点才需要重新着色。实验结果表明，该算法可以在几秒钟内为千万个顶点和一亿个边的图着色。[6]中的工作评估了GPU上图着色的特征，为其他研究人员提供了一些有见地的研究方向。参考文献[13]研究了图着色问题并将其应用于基础设施及在线分析设计空间。

JPL是一种以顶点为中心的着色方案，首先对每一个点赋予了权值，然后在每一轮中用一个颜色去对图中可以着色的点进行着色。如果某一个点有比其权值大的邻接点没有被着色，那么表示该顶点还不能被着色。这种思路对应的串行思路应该是，先用一种颜色着图中所有可以着该颜色的顶点，然后将剩余的未着色顶点构成的图作为下一个颜色着色图的输入。依次执行，直至图被完全着色。

其好处在于每一轮不会有冲突，但是需要轮次多，执行时间长，并行性较差。

cuSPARSE库[6]是由NVIDIA开发的基本线性代数子程序。 cuSPARSE包含一组图处理工具，包括图着色算法，这是迄今为止唯一可用于SIMD架构的公共着色库。

参考文献[1]中的划分队列并利用GPU的流来处理负载不均衡的问题，以及利用扩大空间写入来解决原子操作的问题，以及位操作来处理复杂运算的方法，均对本文有一定的启发。

参考文献[2]在多核体系结构上提出了一种并行着色算法，作者在kokkos库[5]上实现了该算法，该算法可以在Xeon Phi和GPU上使用不同的编译选项执行。实验结果表明，与cuSPARSE相比，该算法可以实现高达1.5倍的加速。

与本文相关性最大的算法是一种为以边为中心的着色方案，后续有二者组合对图进行着色的实验。首先对每个顶点初始化一个颜色号，然后对每一条边开辟一个线程，每个线程的作用是判断其对应的边的两个点的颜色是否相同。当颜色相同时，将编号较大的顶点标记为未着色，并将其对应的颜色号加一。整个更新的过程类似于pageRank，可能存在死循环，为避免这种情况，在最开始对图数据进行了对应的预处理，在第三章再详细介绍。其次，其无法很好的终止，所以设置了一个循环控制的参数，当正确着色的顶点数达到了给定的比例，便终止循环。该算法的伪代码如下所示：

*Algorithm 1 Feluca: A High-Performance Graph Coloring Algorithm*

*Require: Graph, G; fraction*

*Ensure: Graph coloring plan, COLORS; and the colors color\_num*

*used for coloring graph G;*

*1: function RecursionExec(G)*

*2: C(G) ← init\_color\_randomly(G);*

*3: while continue\_ﬂag && colored / Totalvverticesertices ≤ fraction do*

*4: while vj ∈ Vi && i < j do*

*5: if ci == cj then*

*6: update cj ;*

*7: end if*

*8: end while*

*9: if continue\_f laд then*

*10: ParallelSequentialExec(G);*

*11: end if*

*12: end while*

*13: end function*

*14: function ParallelSeqentialExec(G)*

*15: C(G) ← init\_color\_randomly(G);*

*16: while continue\_f laд do*

*17: traverse\_vertex\_dest(row\_ptr[i], colors);*

*18: traverse\_vertex\_src(col\_ptr[i], colors);*

*19: row\_ptr[i] ← colors[i];*

*20: col\_ptr[i] ← colors[j];*

*21: end while*

*22: end function*

这个算法的优点在于：1.边数是足够的，开辟的线程数肯定是足量多的，可以有效地隐藏某些延迟，充分发挥GPU的性能。2.每个线程处理一条边，而所有边的数据是连续的存储于内存中，所以每一次读取的数据都具有较好的连续性。3.核函数控制逻辑简单，不存在分支，且要求分配的空间较少，不会限制硬件上并行的线程数量。

而其对应的缺点在于：1.多轮过后，大多数边的两端顶点颜色均变成了不相同，大量线程并未做实际工作。2.在剩余未正确着色顶点数量较少时，每一轮着色顶点变得极少，收敛速度变慢，需要轮次变多。

而本文设计算法可以很好的与该算法互补，解决上述的问题，提升图着色后半段的性能，从而大幅度提高图着色算法的效率。

## 研究目的和主要内容

根据研究，图着色算法可以分为几类，其中最常使用顺序扩展和迭代。

顺序扩展算法的基本思想是使用算法遍历整个图。采用类似于BFS的执行模式，逐个检查顶点的颜色。这些算法以同步步骤进行，并使用线程处理活跃顶点。同步计算模型的关键属性是同步步骤的数量。具有较少同步步骤的算法可提供更好的性能。在顺序扩展模型中，每个同步步骤可以分为三个阶段。在第一阶段，已着色顶点需要将它们的颜色发送到它们的相邻顶点。在第二阶段，邻居接收关于颜色的消息，然后在第三阶段，邻居执行本地计算。但是，我们从基准测试实验中发现，虽然同步步数很大，但每步中只有少量活动顶点。此外，难以并行运行该执行模型，因为当前迭代的活动顶点只有在已知先前迭代的结果时才能着色。

另一种类型的图着色算法采用迭代执行，其工作方式与PageRank类似。在此着色模型中，每个顶点在开头都分配了一种颜色。然后每个顶点将其颜色值发送给它的邻居。接下来，如果发生颜色冲突（即，相邻顶点具有相同颜色），则相邻顶点更新其颜色。大多数顶点的颜色在第一次迭代中收敛。由于颜色冲突，剩余的一小部分顶点需要很长时间才能收敛到最终颜色。这种长尾现象可以通过实验来证明，我们在NVIDIA Tesla K20m GPU上进行了实验，该GPU配备了5 GB板载内存，2,496个CUDA内核和Red Hat Enterprise Linux服务器6.2版（Linux版本3.10.0 514.el7.x86\_64）。具体情况分析在3.1.4节中。

本文算法思路类似于顺序扩展的思路，是以顶点为中心进行的，通过对于着色方案和预处理的优化，使得效率有了一定的提高。但是其迭代轮数与图的层次数呈现正相关，所以在结构松散的图上有较好的表现。为了使其适用范围更加广泛，将其与以边为中心的算法结合形成了组合算法。先用以边为中心算法快速着色，使得未着色顶点构成的图结构松散，然后预处理，再用本算法收尾，使得效率进一步提高。

在实际的数据中，图有多种类型，有向图与无向图，图的起始顶点索引为1或0，是否允许指向自己的边等。故起始第一步为将图数据初始化，为了避免算法在执行过程中陷入局部死循环，需要将图更改为有向无环图，并以其他的数据形式进行记录。本文做的具体优化在于：

1.预处理图数据并使用CSR和CSC格式来存储以此优化连续访存的问题。

2.将一轮分为两步进行，即为顶点选取合适颜色和处理冲突数，两个过程均是以顶点为中心进行，以此来提高并行度和避免原子操作。

3.组合算法首先使用以边为中心的算法对图中顶点进行快速的随机着色打散图的结构，再使用以顶点为中心的算法进行收尾，解决长尾问题。

在第一种优化方案中，对图数据的预处理和两种数据结构的使用，可以将检查冲突的核函数中的检查所有邻接点是否同色，变成只检查与入度相连的点是否存在颜色冲突，理论上是减少了一半的工作量。

在第二种优化方案中，为顶点选取正确的颜色采用的方法是选取其邻居点中最大的颜色+1，这样可以避免与其邻居点同色，同时不需要对中间状态进行记录，而且将两轮循环变成了一轮。

## 论文结构

本文分为五章。

第一章为绪论，首先是介绍该题目的背景及意义，继而是当前针对该领域已有的研究，如对应的数据结构，编程模型和解决算法等，并着重分析了较为典型的两种算法的优缺点，然后大致介绍了本文的工作内容，最后给出本文结构。

第二章为相关的背景架构知识。首先是GPU的架构以及硬件特点与编程设置的关系；继而是CUDA编程的大致流程，及普遍的优化方案；然后是常用的数据结构的具体说明和分析；最后是分析在GPU下实现高性能图着色算法存在的难点。

第三章为算法的具体思路。首先是对应数据的预处理，然后是其中用到的详细变量的说明；继而是算法的核心函数的大致思路，给出了对应的流程图及详细说明；并分析了两种算法存在的问题，及组合算法带来的优势；再是针对示例给出的大致的流程以及细节部分的说明；最后是点明算法的关键技术及效率提高的实验数据的说明。

第四章为实验配置及运行效率。首先指出了对应的实验环境的具体配置，测试图的数据集及代码的各种参数的设置；其次是组合算法切换时机的分析；继而大致介绍了复现的几种对照算法的思路及测试数据对比的分析；最后是在不同GPU上的测试数据。

第五章为总结。主要是介绍当前工作中可能存在优化的地方，及后续需要扩展的方向。

# 相关背景

为清晰了解后续相关的专业术语以及算法改进思路的理论基础，本章主要对用到的软硬件相关知识和基本优化方法的原因进行详细的说明，主要包括CUDA相关知识，常用的GPU上的大图存储结构CSR及edgelist的特点，以及相关的算法难点分析。

## CUDA相关知识

CUDA是新的通用并行计算架构，拥有独立的开发库和运行环境。它在各种程序中结合了CPU和GPU各自的长处，使得GPU能在更复杂的计算问题上发挥作用，且应用范围更加广泛和通用。

本节主要介绍与CUDA相关的GPU硬件知识，CUDA的编程模型以及基本的优化方法。

### GPU的架构

如图2.1中简易的架构图所示，绿色的是计算单元ALU，黄色的是控制器，橙色的是缓存。可以看出其组成较为简单，控制器小而少，运算器众多，故只能做简单的控制和计算；相似部件多，说明适合并行运行多条相似语句；DRAM的缓存小，而数量多，说明其作用不同于CPU中的缓存。

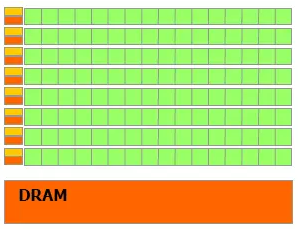


图2.1 GPU简易架构图

如果有很多线程需要访问同一个数据，GPU会将这些字节的访问合并成一个块，然后再去访问存储单元。根据以上特点，GPU适合于计算量大且没有复杂逻辑控制的工作，且最好在每一个核内用到的数据在内存中有一定的连续性的进行存储的，以便利用其特性。除此之外，GPU没有复杂的控制单元，无法处理复杂的语句及逻辑，从而也无法独立工作。其在程序中的用处是由CPU运行程序主干和逻辑运算复杂的地方，然后在需要对大量数据进行相似的处理时，便转到GPU。他们均由自己独立的内存，故开始时的初始化和结束后的结果都需要额外的数据拷贝指令完成。

CUDA编程关注的是软件特征，如图2.2左侧图所示，十分直观的可以看出线程（thread）是被包含在线程块（block）中的。但是，实际运行是在GPU的硬件上的，如图2.2右侧图所示，而硬件上是没有线程块和核函数的概念的。线程都是执行在具体的硬件上的，硬件上的线程数并不恰好是线程块的大小。

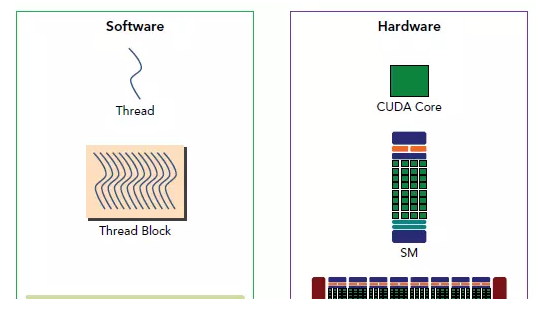


图2.2 CUDA编程与GPU硬件的联系

如图2.3所示，GPU中主要由以下硬件部件构成，它们也是与CUDA编程联系最为紧密的部件，其特征决定了编程时需要注意一些细节来贴合硬件，从而提升整体性能。

SP：最基本的处理单元，流处理器，也称为CUDA内核，类似一个简易CPU。最后在核函数中的具体的指令和任务都是在SP上处理的。在GPU进行并行计算，本质上就是就是很多个流处理器上的线程同时做处理。

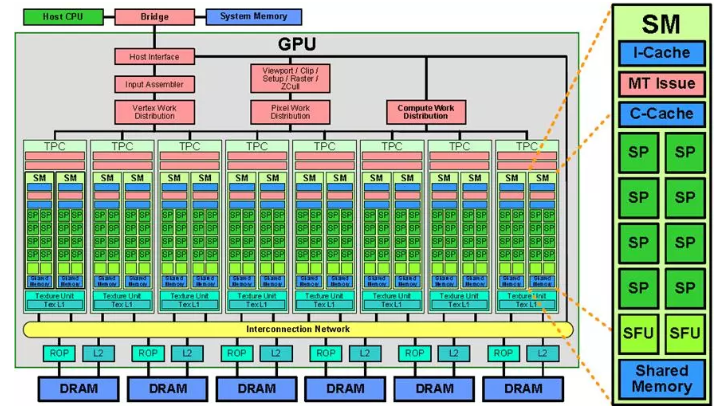


图2.3 GPU具体硬件图

SM：流多处理器，由多个流处理器加上其他的一些额外的资源组成，也叫GPU大核。其他资源如：线程束调度器，寄存器，共享内存等。流多处理器是GPU的核心部件，其中寄存器和共享内存数量相对较少，是流多处理器的稀缺资源。运行时，所有驻留在流多处理器中的线程会瓜分对应的流多处理器上的资源。而没有足够的资源，线程无法运行，故这些资源一定程度上也决定了能够在流多处理器中同时运行的线程的数量，从而每个流多处理器中活跃的线程束被其严格的限制了。

每个流处理器可以执行一个线程，但是实际上并不是一个线程块中的所有的线程能够在同一时刻执行。CUDA把32个线程组成一个线程束，若剩余的线程数不足32，则还是会额外开一个线程束，线程束才是硬件上调度和运行的基本单元。因为上述原因，一个线程块中的线程数目往往会设置为32的整数倍。同一线程束中所有线程并行执行相同的指令，所以同一个线程束中的线程在面对分支语句时，如果判断条件不相同，便会有部分线程出现空转等待的现象。一个线程束需要占用一个流多处理器运行，多个线程束需要轮流进入流多处理器。这个过程由硬件线程束调度表负责调度。

大量的线程可能会被分配到不同的流多处理器，但是同一个线程块中的线程必然在同一个流多处理器中并行（单指令多数据）执行。每个线程拥有它自己独立的程序计数器和状态寄存器，并且用自己的数据执行对应的指令。以上特性也决定的GPU切换任务时不需要类似于CPU进行大量的上下文的准备工作，所以往往有采用运行大量线程，通过线程替换运行来隐藏部分需要等待数据读取的线程的延迟。

### CUDA编程

在进行CUDA编程前，首先需要对以下概念有比较清晰的认识。CPU及对应的系统的内存称为主机。GPU及其内存称为设备。

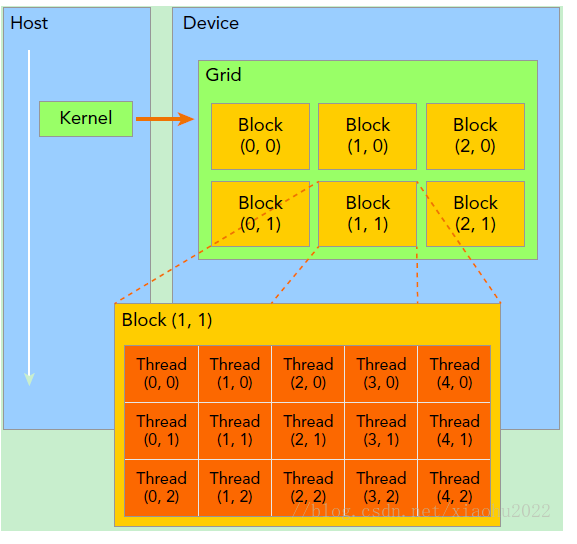


图2.4 线程，线程块与grid

线程(Thread)：执行核函数的最小单位，一般是多个线程同时执行相同语句，仅使用的数据不同。

线程块(Block)：由多个线程组成。各线程块之间是独立的并行的，且无法获取其他线程块的变量信息，他们间的同步和信息交换需要靠核函数的同步来实现。由于硬件上的限制，其块数不可超过65535。

线程格(Grid)：由多个线程块组成。其与线程块之间的关系类似于线程块与线程格之间的关系。他们在组织包含关系时，虽然在硬件上都是一维的，但为了在编程时便于理解且贴合于部分数据结构，可以划分为二维和三维，但实际应用时还是会计算出一维索引。具体布局如图2.4所示。

线程束(warp)：在CUDA架构中，每个线程束由32个线程组成，这个概念是硬件的概念，在编程时不会遇到。但是其特性限制了线程是以32为一个整体同步进行的，所以在一个线程束中的线程最好不要执行的语句有差异。

在实际执行时，为了让每一个线程执行相同的任务，但是处理的却是不同的数据，往往需要对每一个线程进行唯一标识。这个标识类似于将多维（如果将线程块和线程格设置成了多维的话）的展开成一维，其中的threadIdx和blockIdx均是内置变量。其类型是三维的结构体，每维均有一个变量表示该线程在对应方向上的大小。依据上图的结构和其对应的threadIdx和blockIdx的大小，便可以轻松地计算出其id。

核函数（Kernel）：一般是在GPU上执行的函数被称为核函数。该函数与普通函数不同的是，其一般通过标识符\_\_global\_\_修饰，且有<<<参数1,参数2>>>形式的的参数，主要用于说明上述的线程块和线程格的维度大小。还有两个隐藏参数，为共享内存大小和所在流的编号，本文没用到，故不作详细说明。在其执行的内部，可以用语句对同一个线程块中的线程进行同步，但是无法对每一个线程块进行同步。可以说，核函数是线程块同步的方式。其一般不会单独使用，一般是在主机端代码中调用。需要注意的是，核函数中用到的数组或变量是在GPU空间中的，而不是在CPU上的，所以传入的参数必须提前分配好足够的空间，否则在GPU计算时会发生错误。且核函数执行是异步的，即其并不会等待核函数执行完才将控制权返回，而是立即返回。当需要保证线程全部执行完时，需要采用显式或隐式语句进行一次同步。

有了上述认识，代码执行的大致过程如图2.5,执行流程如下：1.分配主存内存，并进行数据初始化；2.分配设备内存，并从主存上将数据拷贝到设备内存上；3.调用CUDA的核函数在设备上完成指定的运算；4.将设备上的运算结果拷贝到主存上；5.释放设备内存和主存上分配的内存。

代码涉及的内存除了基本的全局内存外，还涉及了共享内存（仅在线程块内全局可见，跨块不可见，且分配时默认每块分配一个），寄存器内存等，而且还有用于特殊用途的内存，如纹理内存，常量内存，页锁定内存等，待用时再做详细介绍。

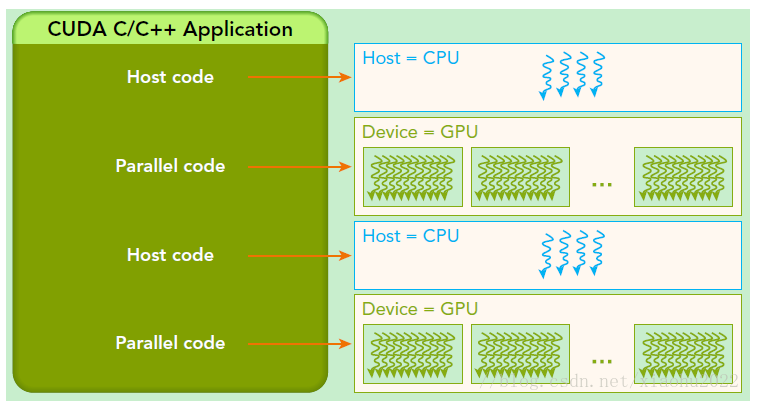


图2.5 CUDA代码执行过程

### CUDA优化技巧

依据上述特点，在实际编程是可以有以下优化手段。

从访存角度而言，首先存储数据的数据结构和算法思路应该匹配，应该访存地址应该尽可能在同一个块中，便于访存合并。其次可以将共享内存的使用进行考虑，将数据量较小的数据从全局内存移动到共享内存，其速度可以有较大提升，但是其大小较小。最后，尽可能避免原子操作，采取的方式可以是，每一个线程处理对应的一块。或者是用空间来换取，首先每一个线程计算自己需要的空间大小，然后再只向自己所属的空间写入。

从编程计算的角度而言，首先尽可能减少核函数中的分支，实在无法减少时，应考虑以线程束为单位去作为判断语句的判断条件。其次应调整语句顺序，将访存，数据传输等语句放在一起，避免多次阻塞。

从线程块大小设置的角度考虑，设置足量的线程有助于隐藏部分因为读取数据而阻塞的延迟，但是过多设置又会消耗额外的寄存器从而影响性能。且块的大小最好设置为32的整数倍，避免线程束的浪费。最后，每一个线程内的寄存器用量，每一个线程块的共享线程用量都有可能限制硬件上的同时并行的线程数量，在使用时应尽可能注意。

## 数据存储

在GPU上进行图的相关操作，图数据的存储形式可能会较大的影响性能。直接采用矩阵，由于顶点数过多，而矩阵大小为顶点数的平方，GPU有限的内存可能无法进行存储。且实际情况下，图是十分稀疏的，较多顶点间是不存在边的，会浪费大量空间。而邻接表之类的数据结构由于其中存在链表与指针，而GPU上是不支持指针操作的，且不利于连续读取，故需要一种新的存储方式。

CSR是GPU下记录图数据比较标准且比较通用的一种，其需要三类数据来完整记录图的数据：边的数值（value），邻接点的列号（colidx），以及每个点行偏移起始量（roffset）。CSR不是三元组，而是整体经过了压缩的编码方式。

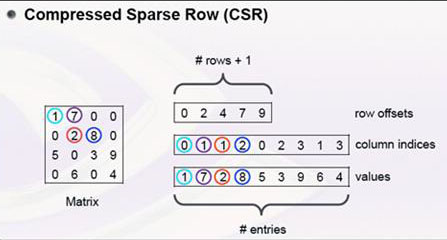


图2.6 CSR实例

其中，每个点的行偏移的起始量数组是一个顶点数+1的数组，每个点以自己的编号为索引读取数值。其表示该顶点第一个邻接点在邻接点列号数组中的偏移量，而roffset[n]- roffset[n-1]就可以表示编号为n的点的边数，所以需要的大小比顶点数多一，便于计算最后一个顶点的边数。最后一个元素的值为边数。

而邻接点的列号（colidx）表示每一个顶点指向的邻接点，如图2.6所示，只有被指向的点才会被记录，且被同一个点指向的点记录在连续的一块，记录位置由roffset数组的值决定，其大小为边数。而边的数值（value）是与之对应的，大小也为边数，用来标注边的额外信息的，比如边长，距离等等。对于图着色中只需要判断两点之间是否有边，故不需要该数组。

CSC是和CSR相对应的一种方式。根据上述介绍，CSR实际是按照行压缩，即有单独的记录出度点的数组。而CSC则是按照列压缩，即有单独记录入度点的数组。以图2.6中矩阵为例，对应的数组值应该为：

Values： [1 5 7 2 6 8 3 9 4]

Row Indices：[0 2 0 1 3 1 2 2 3]

Column Offsets：[0 2 5 7 9]

上述记录方式除了压缩了图数据的大小，便于存储外，还在一定程度上有利于以顶点为中心的算法的数据读取的连续性。因为CSR将一个点的所有邻接出度点都放在一块地方，在用一个线程块处理一个点的所有边时，线程读取的数据在内存中是连续的。

除此之外，对应的还有以边为中心的编程方式，为贴合于其编程特性，创造了edgelist的数据结构。主要包含三个数组，第一个为边的源点数组，第二个为边的终点数组，第三个为边的权值，大小为边数。以图2.6为例，对应数组值为：

Values： [1 5 7 2 6 8 3 9 4]

Src：[0 0 1 1 2 2 2 3 3]

Dst：[0 1 1 2 0 2 3 1 3]

以边为中心编程时，每个边占据一个线程，其优点在于负载均衡且并行度足够，缺点在于每轮运行均要对所有边进行操作，在多数顶点颜色确定时，有效操作比例极其低。

## 难点分析

GPU是一种很有前途的设备，可以加速大规模图上的图着色速度，这与其具有高度的并行性和高内存访问带宽的特点相关。[8]然而，图处理中的固有问题（例如随机存储器访问和工作负载不平衡）使得充分利用GPU的并行计算能力非常具有挑战性。当前已有大量工作来开发新的数据布局模型（邻接矩阵，邻接列表，矢量图，CSR），图编程模型（GAS，BSP），数据布局，内存访问模式，工作负载映射以优化在图上的并行着色。尽管已经有各种尝试，但释放GPU的全部功能以实现高性能图着色仍然是一个巨大的挑战。

在并行架构进行大规模的图着色，主要面临的挑战集中在以下几点：

1. 随机访存的问题。在GPU上，读取数据一般是整合多个需要读取的字节为块，再以块为单位去读取数据，这决定了连续的访存地址能够减少访存次数，获得更高的数据利用率。而图的每个顶点连接的顶点编号可能存在较大的差异，并不是连续。比如一个顶点只有两个邻接点，一个编号为1，一个编号为100000000，而在确认他们的颜色时，需要去读取颜色数组对应的信息，需要读取两个大块。这样就造成了随机访存，访问的地址无法合并为一次，读取内存利用率低下的问题。

图2.7 现实中图DBLP的顶点度数分布图

1. 负载不均衡。在图的并行处理中，有多种并行方式，以顶点为中心，以边为中心，以颜色为中心等。第一种和第三种方案中，常用的做法是采用一个线程对应一个顶点，由于现实图数据都遵循幂律分布，不同的顶点的度数差异巨大，导致不同线程所对应的工作负载差异巨大。如图2.7所示，大部分顶点度数集中在0-100，只有少部分顶点具有极其大的度数，近似于一个指数图像，负载差异巨大。
2. 并行度不足。为充分利用GPU高计算性能和较大吞吐率及线程任务切换代价较小的特点，应在每一阶段均有大量的线程等待执行，以此来隐藏其他开销。如果以颜色为中心去分配任务，对于顶点数在一千万以下，边数在1.6亿以下的每个图所用颜色大致为100-200种，开辟该数量的线程是远不足以发挥GPU的最佳性能的。故对应算法应有能大量线程同时执行的特点，而传统的图着色算法的上一轮的结果为下一轮开始的输入，是不能达到要求的，应重新设计思路。
3. 原子操作影响执行效率。在多个线程并行时，可能有多个线程需要对同一数据进行读写操作，为了避免并行的不确定性导致脏数据，此时需要转化为原子操作。而原子操作是会阻塞其他线程，最差时可能近似于串行，一旦轮次过多，原子操作的开销是无法忍受的。故需要设计避免原子操作的思路来提高算法效率。

图着色中需要考虑的大方向中存在的问题基本为以上几个，而较为细小的比如核函数中减少分支语句，或者使分支判断条件以线程束的大小为单位，调整分块的具体大小等需要在编程中再做详细考虑。

## 本章小结

本章介绍了后续用到的专业名词，以及采取的优化方法及其软硬件架构相关的底层原因，同时说明了图数据在应对对应类型的算法时采取的两种存储结构及优缺点，最后分析了在并行架构下开发高性能的图着色算法时存在的难点。

# 算法实现

本章主要介绍了算法总体思路和详细细节。首先介绍了算法采取的数据预处理方式，继而介绍了用到的变量信息及其意义，然后分析了算法的核心函数的过程，再给出了具体示例的运行过程以及分块和循环控制的细节，并分析了该算法可以优化的地方和融合一种算法后形成的组合算法的大致流程及解决的问题，最后说明了对性能提升较大的三个关键技术。

## 算法思路

### 数据预处理

由于图数据类型较多，且在细节设置上存在差异。在程序运行参数表中给出了对应的详细参数设置，并给予了一定的预处理。参数设置包括，是否为有向图，顶点索引的开始编号等。

预处理主要是因为当基于递归的着色算法在带有环形路径的图上工作时，算法是沿着边的方向运行（不论是以顶点为中心还是以边为中心都是依据邻接点颜色来给定颜色），因此很容易陷入无限循环的着色中。为了解决这个问题，在本算法中开发了一种方法来消除有向图中的循环路径。在该方法中，本算法在着色顶点vi之前检查所有顶点vj∈Vi（即，顶点vi的邻接点集合），并且如果i> j，则将边<vi，vj>改变为<vj，vi>。例如，图3.1中的边<v4，v1>在图3.2中变为<v1，v4>。边方向的改变不影响颜色的限制关系，故不影响最后的着色结果。例如，图3.1和图3.2中的这两种着色方案被认为是相同的。

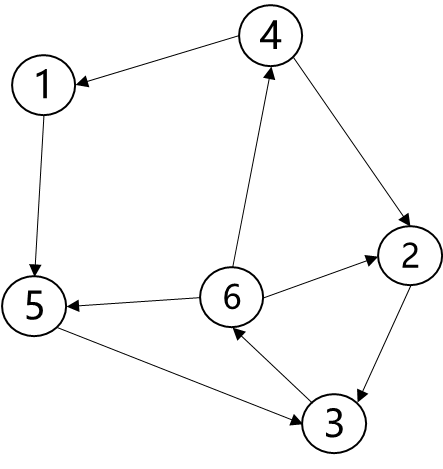


图3.1 图结构调整前

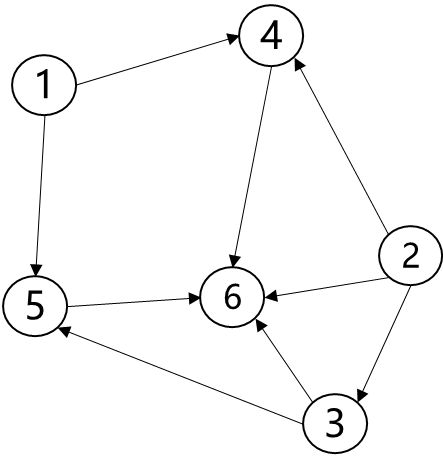


图3.2 图结构调整后

进行预处理后，对应的CSR数据中只存了每个顶点出度边对应的点，而损失了入度边对应的顶点，这部分数据需要在第一个着色核函数中使用，所以还需要添加图的CSC数据。

最后，如果需要使用以边为中心的着色函数，可以增加在进行预处理后的图的edgelist数据表。

### 详细变量表

算法中用到的详细变量见表3.1，命名规范均按照在GPU上的变量有dev，在CPU上的相同变量名字相同但是没有对应的dev前缀的约定。

表3.1 算法详细变量

| 变量名 | 空间大小 | 所在位置 |
| --- | --- | --- |
| remain\_ver\_sum | 剩余顶点数\*sizeof(int） | GPU |
| dev\_remain\_ver | remain\_ver\_sum\*sizeof(int） | GPU |
| dev\_col | sizeof(int) \* n\_edges | GPU |
| dev\_row\_ptr | sizeof(int) \* (n\_vertices+1) | GPU |
| dev\_row | sizeof(int) \* n\_edges | GPU |
| dev\_col\_ptr | sizeof(int) \* (n\_vertices+1) | GPU |
| dev\_undone | sizeof(int) \* n\_vertices | GPU |
| dev\_colors | sizeof(int) \* n\_vertices | GPU |
| dev\_changed | sizeof(bool) | GPU |
| dev\_color\_unvalid | sizeof(bool) \* n\_vertices | GPU |
| changed | sizeof(bool) | CPU |
| colors | sizeof(int) \* n\_vertices | CPU |
| num\_undone | sizeof(int) | CPU |
| undone | sizeof(int) \* n\_vertices | CPU |

上述变量中，dev\_col和dev\_row\_ptr均是图进行预处理后的CSR的数据，而dev\_row和dev\_col\_ptr是图的CSC的数据。

dev\_remain\_ver和remain\_ver\_sum在算法单独使用时，表示图顶点数和顶点编号数组。如果是放在以边为中心的算法后进行执行，表示剩余顶点数和剩余顶点的编号数组。

dev\_undone和undone表示该顶点的颜色是否已经确定，为1表示没有确定，为0表示确定，这样可以避免在多数顶点着色后，在已着色顶点上浪费时间。num\_undone用于记录还有多少剩余顶点没有被着色。而dev\_color\_unvalid用于判断该轮为对应顶点选取的颜色是否符合要求，主要在每轮第二个检查的核函数中使用。

dev\_colors和colors用于记录每个顶点被着色的颜色编号。dev\_changed和changed用于记录每一轮中是否有顶点的颜色被改变，如果某一轮没有任一顶点的颜色发生改变，说明已经稳定，可结束循环；而存在顶点的颜色改变，则继续循环。如果使用了以边为中心的算法，需增加相应的edgelist数据。

以上变量应该尽可能在同一块进行初始化（理论上可以减少分配空间的时间），且对于有顺序的大规模的变量，比如每个顶点的颜色数组，可以使用核函数并行处理，每个线程对一个点的对应变量进行初始化。对于大小较小而无顺序的数组应采用基本的赋值函数，因为开辟线程也会产生开销。

### 着色与修正函数

算法的整体思路为，在每一轮内有依次有两个并行的核函数，第一个负责顶点着色，第二个负责错误颜色的修正。第二个核函数必须等待第一个核函数执行完毕才能执行，每一轮均需要等待上一轮执行完毕才能开始执行。

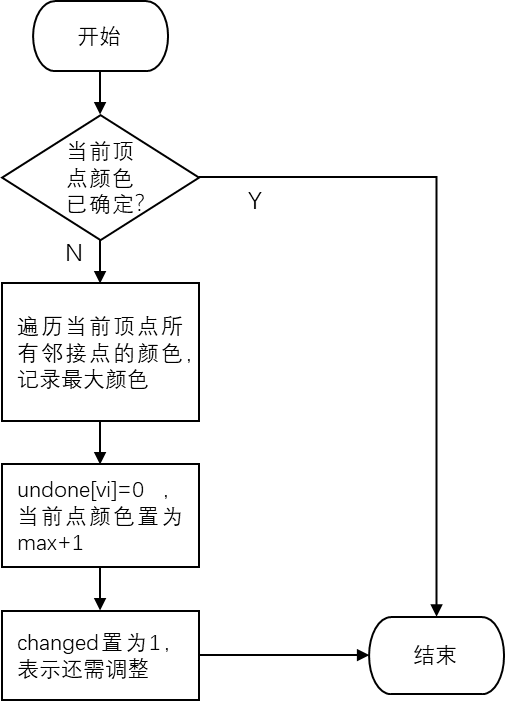


图3.3 第一个核函数

第一个核函数如图3.3所示，是以顶点为中心设计的，每一个线程处理一个顶点，主要目的是寻找这个顶点对应的颜色。首先是判断该顶点是否已经被正确着色（有对应的标志变量），如果已经被正确着色，便结束该顶点的处理；如果没有，便继续。继而选择一个合适的颜色，这个过程并不要求颜色一定正确，但是要求与目前已着色的邻接点不要有冲突，且尽可能没有对两个未着色顶点分配同种颜色。

其中一种颜色选取方法是，用一个大小为图的最大出度的数组，记录每一个邻接点的颜色，然后再循环遍历这个数组，找到第一个可用最小编号的颜色（这个过程可能循环多次）。该方法存在的问题是，在图较大的时候，每一个线程开辟对应大小的数组可能造成空间不够及并行度减少；其次是，在已使用的颜色编号较多时，可能循环多次造成较大的时间开销。

当前采用的方法是，直接选择邻接点中颜色编号最大的数字加一作为当前顶点的颜色。其好处在于只需要一个整型大小的空间，且只需要一次循环即可，且不会与邻接点的已有颜色冲突。但可能存在更小的可使用的颜色编号，所以会使得最后使用的总颜色数有所增加。

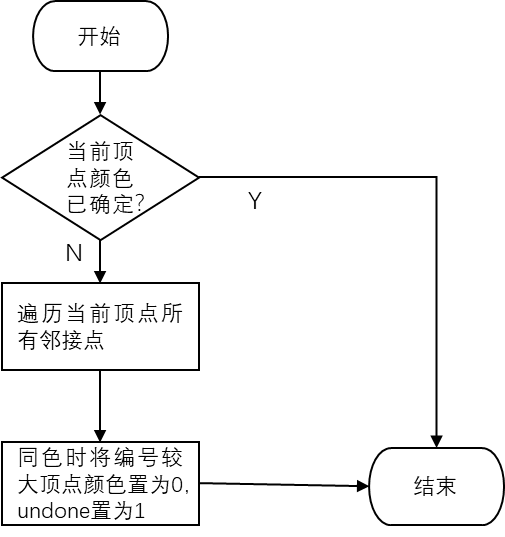


图3.4 第二个核函数

第二个核函数如图3.4所示，也是以顶点为中心设计的，每一个线程处理一个顶点，主要作用是验证当前点是否被正确着色。第一步依然是判断该顶点是否已经被正确着色（有对应的标志变量），如果已经被正确着色，便结束该顶点的处理；如果没有，便继续。下一步，由于已经对图进行了预处理，所以这一步只需要判断出度表中是否有与自己同色的顶点，如果同色，便将编号较大的点颜色置为0，且undone置为1，表示这个点的颜色还不合规范，下一轮中仍然需要对其着色。这个过程只用了CSR的数据而没有用CSC的数据。且如果有顶点的数据被修改，那么便将changed置为1，表示下一轮还需要进行循环。

依次进行，直至所有顶点均被正确着色，算法结束。

### 组合算法

在1.2节中介绍的以边为中心的算法，由于数据结构和设计思路，可以较好地解决负载不均衡和随机访存的问题，且每一轮运行需要的时间极短。而存在的问题是多轮迭代后，未着色顶点相关的边较少，导致实际起作用的线程较少。如图3.5所示，在迭代10轮以后，冲突顶点比例（未正确着色顶点）升高至80%以上，即起到实际作用的线程极少。

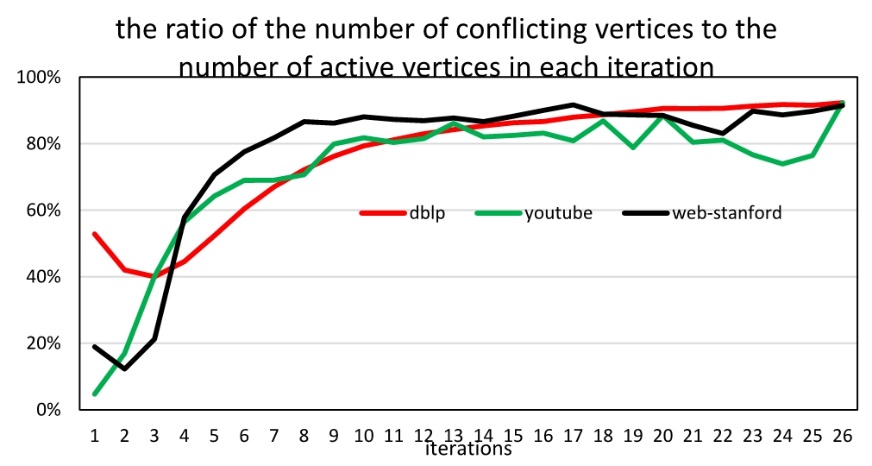


图3.5 每轮冲突顶点数

其次是算法随机性较强，在最后每轮能着色的顶点数极少，可能造成剩余1%的顶点需要60%的时间的情况。如图3.6所示，在迭代5轮后，正确着色顶点数极少，而将剩余不多的顶点完全着色需要20轮迭代，即长尾问题。

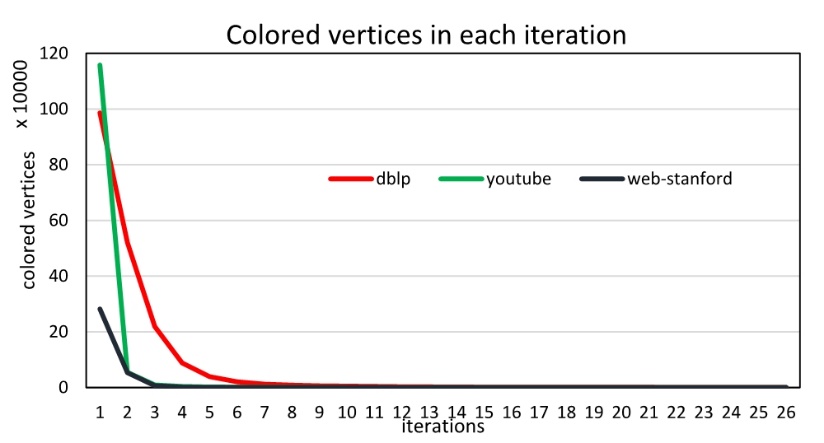


图3.6 每轮着色顶点数

先用以边为中心的算法迭代部分顶点后，再使用本算法处理后续顶点可以较好地解决长尾问题。经过实验与分析，本算法迭代轮次和迭代时间与图的BFS层次多少成正相关。通过以边为中心算法的预先处理，剩余未着色顶点的编号较为随机，组成的图结构松散，可以使得BFS层次深度相对于原图数据有较大的下降，从而使的本算法迭代轮次减少，时间效率提高。同时，使用本算法处理后续剩余未着色的点，也较好地解决以边为中心算法对于剩余顶点处理效率低的问题。

在结合时，切换算法时，也需要进行预处理。即构造未着色顶点编号的队列和相关变量的初始化，再用以顶点为中心的算法处理，可以避免线程浪费。组合算法需要选择合适的迭代时机以达到最好的性能，一般而言，结构紧凑，BFS层次较多的图切换时机较为靠后；结构松散，BFS层次较少的图切换时机较为靠前。该部分在第四章结合实验详细说明。

## 算法分析

### 运行过程示例

为清晰了解算法的整体过程，现对图3.7示例数据给予对应的操作。可以看出，图中的数据已经做过了处理，均是由小编号点指向大编号顶点，这样可以避免染色的震荡。其次模拟操作过程无法真实的体现GPU的并行过程，因为GPU的并行，每一个线程的执行顺序是随机的，存在多种可能的顺序；而模拟的并行过程是按照线程编号依次执行的，按照实际效果来说，该模拟过程实际是最差的效果的模拟过程，所以只能大致作为理解的参考，不代表实际运行过程。

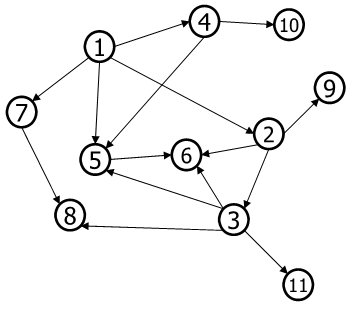


图3.7 着色示例图

执行过程如表3.2所示。首先第一轮为数据的初始化，所以顶点颜色均为0，对应的undone值和valid值均为1。而第1.5轮是为了更好的理解核函数，给出第一个核函数执行后的样子。

表3.2 算法运行过程

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 轮次 | 顶点号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 1 | color | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| undone | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| unvalid | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1.5 | color | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| undone | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| unvalid | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | color | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| undone | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| unvalid | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 3 | color |  | 2 |  | 2 | 0 | 0 | 2 | 0 |  |  | 2 |
| undone |  | 0 |  | 0 | 1 | 1 | 0 | 1 |  |  | 0 |
| unvalid |  | 0 |  | 0 | 1 | 1 | 0 | 1 |  |  | 0 |
| 4 | color |  |  |  |  | 3 | 0 |  | 3 |  |  |  |
| undone |  |  |  |  | 0 | 1 |  | 0 |  |  |  |
| unvalid |  |  |  |  | 0 | 1 |  | 0 |  |  |  |
| 5 | color |  |  |  |  |  | 4 |  |  |  |  |  |
| undone |  |  |  |  |  | 0 |  |  |  |  |  |
| unvalid |  |  |  |  |  | 0 |  |  |  |  |  |

第1.5轮实际上是为顶点选择一个颜色，这个颜色可以是错误的，但是会将其unvalid置为1，且错误的着色结果是会在第二个核函数中被修正的。所以，到了第二轮，对于有边而且颜色相同的两个点，会将编号较大的点的颜色重新置为0，其余变量置为1，表明它没有被正确着色；而将编号较小的点颜色固定，其余变量置为0，表示其颜色已经固定。最后，每轮中对应变量没有被修改的点，表明其颜色不会有任何问题，其unvalid和undone均会变为0，颜色编号也会确定，在后续的染色过程中也不会再发生改变。

表3.3 示例着色结果

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 顶点号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 颜色 | 1 | 2 | 1 | 2 | 3 | 4 | 2 | 3 | 1 | 1 | 2 |

然后依次进行对应的操作，在第五轮便完成了整个着色过程，而着色结果如表3.3所示，将结果进行对应的验证，发现结果是正确的。且总共使用的颜色数也只有4种，总的迭代轮次也十分少，所以总体而言，着色效率较好。

### 算法细节分析

核函数分块时，每个线程块中的线程数目（nt）应该是32的整数倍，其次为了提高并行度，应在允许的范围内适当的大些，所以定为256，而线程网格的大小只受到硬件限制，所以设置为int nb=(remain\_ver\_sum+nt-1)/nt;nb=nb>MAX\_GRID\_SIZE?MAX\_GRID\_SIZE:nb。同时可能存在顶点数过多，一个线程处理一个顶点不足以完成的情况，所以每个线程都加了一个循环，处理完当前顶点后，再处理下一个编号为当前编号加上线程总数的顶点。

如果该算法用在以边的中心的算法的后面，由于前一个算法已经对部分顶点进行了对应的着色，为避免重复处理，输入的图应该是剩余未着色顶点组成的图。所以可以采用串行检测顶点是否被着色并加以记录，然后采用一个并行核函数，对后续算法会用到的变量进行初始化。由实验测试可得，初始化仅需要2ms，消耗的时间远小于其所带来的加速收益。

着色结果的正确性验证是通过对于每一个顶点，观察与其相连的点是否与其同色来判断的，该判断方法完美贴合于图着色的定义。而着色数则是通过一个数组记录所有被使用的颜色色号，然后根据这个数组大小给出的，并不是直接根据颜色的最大编号来的，这样可以避免未使用的颜色编号被计数。程序的计时是调用的linux下的时间函数，可以精确到微秒级别，并且每一个核函数的计时必须在核函数执行完毕后有一个显式或是隐式的同步后，再调用时间函数，因为核函数的调用是异步的，不同步会得到错误的过短的执行时间。

最后，实验整体是在NVIDIA Tesla K20m GPU上进行的，但是部分大图，由于数据量过大，而该GPU的内存大小不足，所以两个最大图的算法测试是在内存更大的NVIDIA P100上进行的（实验数据均已指明）。

## 算法关键技术

### 顺序扩展

预处理将图的结构变为了只存在从小编号顶点指向大编号顶点的边的有向无环图，这样做的好处在于每个顶点着色实际有了优先级，编号越小，优先级越高，且该预处理开销极其小。

在3.2节的实例操作中可以看出，着色过程大致是按照BFS的顺序进行的，第一轮正确着色的点为BFS第一层的点，第二轮正确着色的点为BFS第二层的点，依此类推。原因在于，第一层的点优先级最高，不会受其他点的影响，故第一次便可确定。而第二轮时，比第二层优先级高的点均已被着色，故此时第二层的点可以从未被第一层使用的颜色中选取颜色并确定，以此类推。

理论上，图有多少层便需要多少轮，结构越紧密的图需要轮次越多，耗时越长；结构越稀疏的图需要轮次越少，耗时越短，实际情况也与之相符。但是在实际操作的过程中，算法实际运行的轮次数小于层数，因为进行颜色选取时，每一轮并不是只选择一个颜色在着色，部分在其它层的顶点可能也被正确着色了，从而加快了着色速度。

其优势在于不会存在反复着色和循环着色，着色过程近似于BFS的顺序着色，且迭代轮次在可以接受的范围内。该预处理同时也保证了算法必然会终止，因为有向无环图的BFS层数是有限的，最大迭代次数即为该层数。

预处理带来的另一个好处在于，在修正时，只需要查看出度边对应的点，而不需要查看入度边对应的点，理论上可以减少一半的工作量。

### 避免原子操作

动态冲突列表是用于存储要在后续迭代或循环中处理的顶点（或边）的数组，广泛用于基于顶点和基于边的图处理模型。在图着色中，动态冲突列表不仅用于维护接下来的工作顶点，还用于将候选颜色存储在颜色数组中。典型的着色算法一般通过遍历颜色数组来为当前顶点选择合适的颜色。原子操作广泛用于管理对GPU上的公共数据的同时访问。由于GPU中的SIMD执行模型，会导致大量线程同时访问颜色数组。当频繁访问时，原子操作可以会造成繁重的开销。此外，原子操作可能会导致冲突列表分散，即连续的顶点在存储位置上相隔很远。

在以前的工作中已经考虑过消除GPU上用于维护着色队列的原子操作[7,30]。参考文献[7]是关于多核架构（包括Xeon Phi和GPU）的代表性着色图。在他们的工作中，作者定义了两个队列。一个用于记录活动顶点v的禁用颜色，而另一个用于选择最小的可用颜色。这两个数组分别命名为VFORBIDDEN和ASSIGNCOLORS。作者使用原子操作来维护这两个数组。这种方法是无效的，因为它无法完全避免原子操作，并且VFORBIDDEN需要比传统遍历算法更多的内存空间。示例图如图3.7 着色示例图。图3.8显示了GPU上具有原子操作的典型颜色数组的示例。正如我们在图3.8中所示，当算法为顶点1和5着色时，处理这两个顶点的两个线程需要同时访问颜色数组。为确保两个线程都能获得正确的颜色，需要进行原子操作。

有几种方法可以解决颜色冲突，例如随机选择另一种新颜色或在发生冲突时指定第一种可用颜色。但是这些方法可能会导致最终结果中的颜色总数很大。选择第一种可用颜色是最广泛使用的方法。在该方法中，算法沿着颜色队列搜索以获得第一中可用颜色（即，颜色不用于先前的顶点）。由于GPU内核并行运行大量线程，因此需要原子操作或锁定以确保每个线程都能获得正确的颜色。这将大大减慢着色过程。

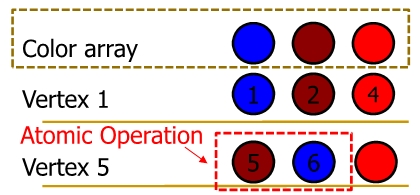


图3.8 原子操作示例

在本算法中，提出了一种带有修正核函数的颜色选择方案，用于为冲突顶点选择合适的颜色。该方案可以完全避免原子操作。接下来我们将详细介绍我们的颜色选择方案。

算法采取的步骤是先为对应顶点选择一个颜色，且这个选择颜色过程是并行的，如3.2.2节中给出的示例一样，相邻顶点间是存在选择相同颜色的情况。但是这一步的主要目的是，为还未着色的顶点分配一个不会跟已着色邻接点同色的颜色，且该颜色尽可能不和其他未着色邻接点分配同色，但是不要求完全正确。其中存在冲突的必然是一小部分顶点（在硬件上一起运行的线程才会冲突），而第二个核函数的工作就是针对该情况进行修正。

处理存在相同颜色的相邻顶点的方法是，将编号较小的顶点颜色固定下来，状态置为已着色；而编号较大的顶点的颜色置为0，状态置为未着色，以便下一轮再次对其进行着色。

按照该方法，多次迭代，前面论述了该算法必然会终止，且在整个过程中完全没有原子操作，运行在顶点上的线程是完全并行的。

### 增加并行度

将着色和修正分解为两个阶段的核函数，除了可以避免原子操作，也可以使得每一轮未着色的顶点均能使用一个线程来完成自身更新，提高并行度。配合新的颜色选取方案，可以使得每个周期内的有效指令和每轮迭代的正确着色顶点比例大大升高。

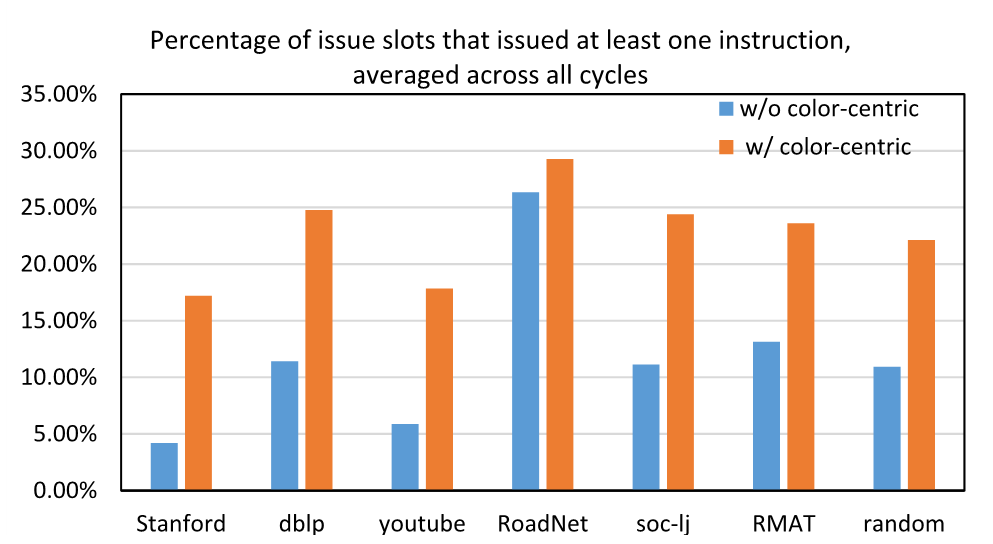


图3.9 每周期有指令发布的发布槽比例

图3.9显示了发出至少一条指令的发布槽的所占百分比，在所有周期中取平均值。结果表明，有颜色优化选取方案的范例可以将性能提高111％（在RoadNet数据集上）-410％（在web-Stanford数据集上），这表明通过使用有颜色优化选取方案，在每次迭代中都执行了更多的指令。

图3.10绘制了平均每个周期活跃的平均warp数。图3.10显示，在RoadNet数据集上，如果没有颜色优化选取方案的范例，每个周期中的平均活跃warp数不超过2.2，而有颜色优化选取方案的范例对应的值则增加到3.12。在web-stanford数据集上，有颜色优化选取方案的范例可以将每个周期中的平均活跃warp数提高至原来的4.9倍。这些实验表明，有颜色优化选取方案的范例可以改善每个执行周期中的活跃warp数，这意味着通过使用有颜色优化选取方案的范例，每轮中有了更多的活跃线程。

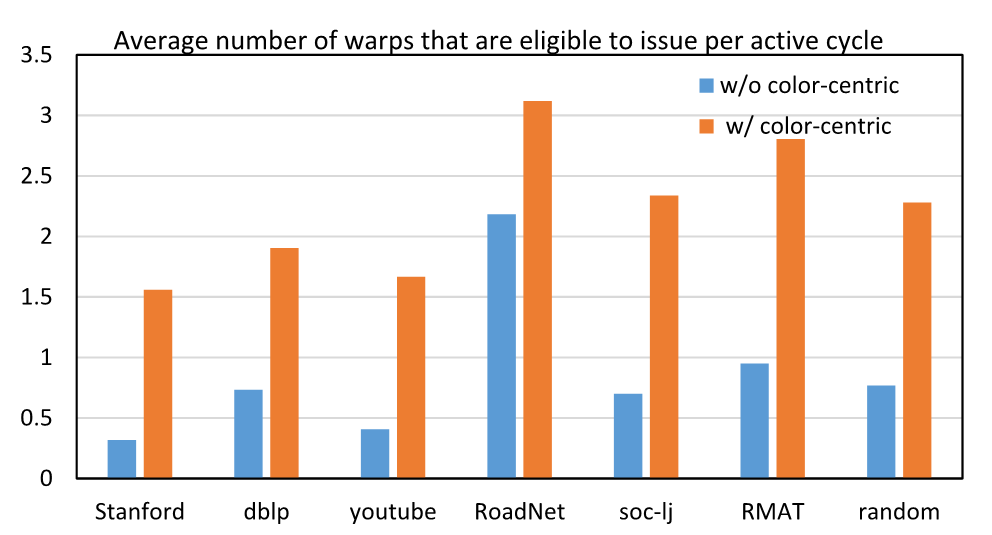


图3.10 每周期平均活跃warp数

除此之外，该方法还可以使得每轮中正确着色顶点比例变高，从而使得迭代轮次减少，着色过程能够更快完成。在下面的实验中，我们通过使用和不使用优化的颜色选取方案两种方法来实现顺序扩展算法。该实验旨在显示每次迭代中的冲突顶点数与活动顶点数之比。实验结果如图3.11所示。图中显示，如果没有优化的颜色选取方案的范例，测试的三个数据集，web-Stanford，youtube和RandomGraph，至少需要24次迭代才能收敛。在有颜色优化选取方案的范例中，RandomGraph在第7次迭代时便已收敛，而其他两个数据集在便已第11次迭代时收敛。

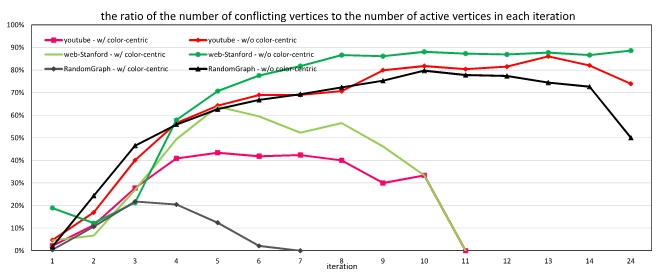


图3.11 收敛次数与颜色选择

图3.11还显示，对于有颜色优化选取方案的范例，对于youtube和RandomGraph，每次迭代中的冲突顶点数与活动顶点数之比不超过45％，而在没有颜色优化选取方案的范例中，冲突比例可以增加到80％-87％。有颜色优化选取方案的范例可以避免在这两个数据集中的大约50％的冲突。在第10次迭代中，没有颜色优化选取方案的范例中，web-Stanford的冲突比例增加到87％，而有颜色优化选取方案的范例中，冲突比例不超过63％。且在每一个图数据上的迭代轮次有了明显的减少，例如在youtube上，没有颜色优化选取方案的范例迭代轮次为24次，而有颜色优化选取方案的范例的迭代轮次减少到了10次。

综上所述，该算法相比于以前的算法，在每个周期内的有效并行指令增多，活跃的warp数增加，即每轮运行时间会有所减少。且每轮迭代中着色存在冲突的顶点比例减少，使得迭代轮次变少，总运行时间减少。

除了以上优化，使用以边为中心的算法后，并没有直接在剩余数据上进行下一算法的操作，而是采用了预处理，将剩余顶点进行集合，并且给出了新的用于标记其是否确定着色的变量。这样不仅保证了后续的操作过程中有较好的空间连续性，也保证了避免在已着色的顶点上浪费线程资源。且以边为中心的算法采用的是edgelist的数据结构，该结构可以完全贴合于该算法，保证负载的均衡性和访问存储空间的连续性。

颜色选取方法也相应的提升了较高的性能，避免了多轮循环，多轮访存，也避免了过多开辟线程的私有空间造成空间不足，从而影响硬件上的并行度。

## 本章小结

本章给出了算法对于原始数据的预处理过程，详细描述了该算法运行过程中的各个细节以及总体的执行思路，并针对算法主体思路的两个函数给出了具体的流程图。且为了清晰分析算法的详细步骤，针对示例演示了算法的详细过程并制定了表格。继而分析了该算法可提升效率相关的点，融合优化了一种算法，形成了组合算法，解决了对应的问题，提升了性能。同时在最后分析了对于性能提升较大的三个关键技术，并解释了这些优化方法带来性能提升的实验数据和性能提升的主要原因。

# 性能评估

在本节中，我们将介绍本算法中数据对比使用的数据集和具体的测试环境，并设计了不同的实验来全面评估实验结果，分析了组合算法在不同特点的图数据上的切换时机，最后将本算法与随机幂率图上最先进的图着色技术进行了各方面详细的比较。

## 实验设置

### 数据格式

实验中，图的类型有有向图和无向图两种。其中（u，v）表示顶点u和v之间的无向边，而<u，v>表示从u到v的有向边。图存储在纯文本文件中，其中每个顶点具有唯ID，并且存有向或无向边的数据，每行数据表示一条边。采取了3.1节中的预处理方式，将所有的边均转化为了从小编号顶点指向大编号顶点的有向边，将无向图转换为了有向无环图。

### 数据集

表4.1 实验所用的图的数据集

|  |  |  |  |
| --- | --- | --- | --- |
| 数据集名 | 顶点数 | 边数 | 是否为有向图 |
| web-stanford | 281903 | 2312497 | 有向图 |
| dblp | 986207 | 6707236 | 无向图 |
| youtube | 1157828 | 2987624 | 无向图 |
| RoadNet-CA | 1971282 | 5533214 | 无向图 |
| Wiki-Talk | 2394385 | 5021410 | 无向图 |
| soc-LiveJournal | 4847571 | 68993773 | 有向图 |
| RMAT16-2 | 9999993 | 160000000 | 无向图 |
| Random-graph | 41652230 | 100000000 | 无向图 |
| twitter-2010 | 41652230 | 1468365182 | 有向图 |
| webbase-2001 | 118142155 | 1019903190 | 有向图 |

我们对总共10个不同的图，包括8个真实图和2个合成图进行了实验测试，如表4.1所示。图中的顶点数量范围为0.3亿到1.18亿，边的数量范围从2.3百万到14.6亿。所有图中的顶点度数范围为2到。图RMAT16-2和图random-graph是使用PaRMAT [3]生成的，他们具有随机分布的特点。其他八个图数据是从实际问题中提取的。twitter-2010和webbase-2001在Web算法实验室（LAW）[3]中共享，其余图数据来自斯坦福网络分析项目（SNAP）[5]。

### 测试环境

第4.2-4.4节中介绍的实验是在NVIDIA Tesla K20m GPU上进行的，这是一款基于Tesla架构的GPU，具有5 GB板载内存和2,496个CUDA内核。

该GPU与配备2个Intel（R）Xeon（R）E5-2670 CPU的主机配合使用，每个CPU为2.60 GHz，内存为8 GB。主机正在运行RedHat OS版本4.4.5-6。 该算法用C++和CUDA 9.0实现，使用了“-arch = sm35”计算兼容性文件。 CPU\_greedy着色算法在上述主机中进行测试。JPL和cuSPARSE算法[6]的实验结果是基于CUDA和C++实现的，在相同的测试环境中重现。

## 切换执行阶段的时间

本文算法单独运行时，具有可观的效率，但在与以边为中心的算法结合时，先用先用以边为中心的算法处理部分数据，可以得到更好的效率，本节着重讨论二者切换时间与最终效率之间的关系，而单独的本算法运行时间由4.3节给出。

为了得到参数fraction的合适值，该参数用于控制算法的执行何时从递归阶段切换到顺序扩展阶段，我们为具有不同fraction值的不同数据集设计了一组实验。如以下图组所示，本文选取了四个个有代表性的图数据在各个切换阶段运行时的详细信息。下图中的左侧y轴是以毫秒为单位的着色时间，而右侧的y轴是颜色的数量。带有灰色菱形点的线表示算法的总着色时间，而黑色的方块线和绿色的圆形线分别表示递归和连续扩展阶段的着色时间。颜色数由带有三角形点的线表示。

我们可以从组图中得出以下观察结果。

（1）在fraction具有较小的值时，顺序扩展阶段耗时较大，这意味着在递归阶段中只有很少的顶点被着色。根据定义，是递归阶段的着色率。当λ≤fraction= 0.5时，意味着在递归阶段有半数的顶点被着色，而另一半在连续扩展阶段被着色。上图显示出，在所有幂律图中，递归阶段的执行时间均远小于顺序扩展阶段的执行时间，这意味着从头到尾跑完，递归算法比顺序扩展算法快得多。每一轮运算，递归算法比顺序扩展算法快得多。相反，递归方法需要更多的时间和一个大的fraction值，这意味着在递归阶段结束时图结构会更加松散。

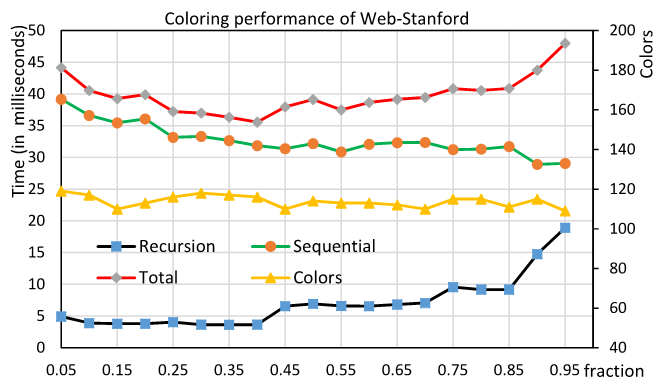


图4.1 切换时间-Web-Stanford

（2）该组合算法可以在幂律图和具有少量颜色的随机图上获得更加良好的性能。

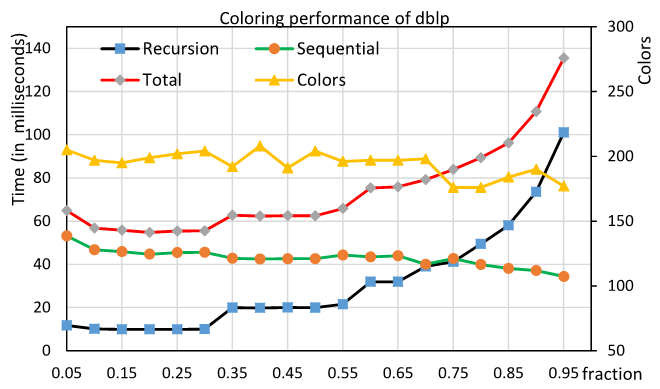


图4.2 切换时间-dblp

（3）组图显示组合算法的执行时间是一个凸函数。因此，总体而言，当执行时间函数的导数接近0时，组合算法可以达到最佳着色时间。执行时间函数的导数可以近似为。可以假设两次连续迭代的执行时间几乎相同，然后，当活跃顶点的数量与两次连续迭代中的冲突顶点的数量相同时，组合算法从递归阶段切换到顺序扩展阶段。

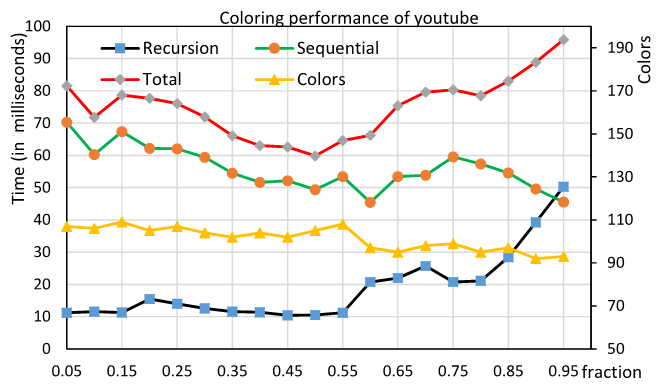


图4.3 切换时间-youtube

同时，可以看出在每个图数据中的切换时机和总时间变化存在一定差异，这主要是由图的结构决定的。图的结构越松散，层次越少，fraction便越小；图的结构越紧密，层次越多，fraction便越大。

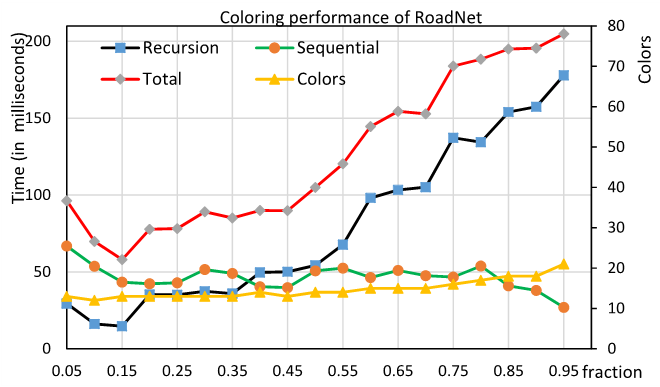


图4.4 切换时间-RodaNet

递归方法在后来的迭代中变得不那么有效，这被称为长尾问题。由于组合算法按照边的方向将颜色分配给冲突顶点，因此大多数冲突顶点可以在第一次迭代中找到合适的颜色。但是，在大多数顶点找到合适的颜色后，这些已着色顶点将对剩余顶点的颜色产生影响。这会导致少量剩余顶点在以后的迭代中重复更改颜色，从而拖慢着色进度。这就是为什么需要在当满足从所述的条件时，组合算法从递归阶段切换到顺序扩展阶段。

## 与最新的技术比较

我们将组合算法与最近的一些研究进行了比较，例如cuSPARSE，JPL[6]和贪婪着色算法基于CPU的实现。下表显示了所有十个图的执行时间和颜色数。每个图中绘制的性能值是基于GPU的解决方案（组合算法，cuSparse和JPL）的5次独立运行的平均值或基于CPU的10次运行的平均值。实验结果显示组合算法最高相比于基于CPU的Greedy着色算法达到125.81倍的加速，相较于JPL算法加速71.07倍，比最先进的cuSPARSE算法加速96.73倍。作为参考，最新的一种算法[7]仅比cuSPARSE加速了1.5倍。

表 4.2 算法时间及着色数对比

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Datasets | CPU\_Greedy | | 本文与递归的组合算法 | | JPL | |
| Time | Color | Time | Color | Time | Color |
| web-Stanford | 103.955 | 149 | 28.81 | 92 | 1121.79 | 169 |
| dblp | 322.065 | 119 | 42.388 | 139 | 787.762 | 121 |
| youtube | 216.808 | 43 | 58.777 | 57 | 1409.49 | 262 |
| RoadNet | 145.619 | 5 | 12.524 | 6 | 806.787 | 13 |
| Wiki-Talk | 774.658 | 97 | 122.464 | 95 | 8703.57 | 489 |
| soc-LiveJournal | 4805.73 | 328 | 448.574 | 551 | 11000.5 | 646 |
| RMAT16\_2 | 8965.46 | 75 | 485.55 | 410 | 18774.9 | 316 |
| random-graph | 9076.56 | 94 | 353.643 | 86 | 17264.3 | 334 |
| twitter-2010 | 83796.8 | 918 | 5040.66 | 947 | null | null |
| webbase-2001 | 343404 | 1507 | 2729.5 | 1559 | null | null |
| Feluca Speedup | 3.61× – 125.81× | | null | | 18.58× – 71.07× | |



如表 4.2显示，组合算法在所有十个数据集的运行时的效率方面优于所有其他竞争对手。除了cuSPARSE之外，所有这些算法都可以生成正确的着色结果，而仅cuSPARSE是一种不完整的着色算法。例如，在soc-LiveJournal中的4,847,571个顶点中，在webbase-2001的118,142,155个顶点中和twitter-2010的41,652,230个顶点中，cuSPARSE分别将2,437,231和22,396,212和104,173,619个顶点分配了相同的颜色。从另一个角度来看，当发生冲突时，cuSPARSE不会为顶点选择一种正确的颜色。twitter-2010的41,652,230个顶点在组合算法中用947种颜色着色，而cuSPARSE仅使用了917种颜色正确着色了19,256,018（所有顶点的46％）个顶点，并将剩余的22,396,212个顶点分配给了存在冲突的颜色。上表显示cuSPARE可以使用比组合算法更少的颜色为soc-LiveJournal，twitter-2010和webbase-2001数据集着色。但cuSPARSE的颜色数量较少也是由于其解决方案的性质不完整。

本文算法单独使用时，部分图的运行时间和着色数基本与组合算法一致，因为在这些图上，图结构稀疏，较为适合使用该顺序扩展算法。全部使用顺序扩展算法反而可以得到更好的性能，且不会有长尾问题。其次，在大部分图上，本算法存在的问题为，顺序扩展轮次多，用时较长，且需要的着色数基本为其他算法的1.5倍。所以需要一定的组合，来发挥其最好的效率且使用较少的颜色数。

如4.1节所述，组合算法仅关注当前顶点及其邻接点，不重复搜索整个颜色数组以找出当前顶点的可用颜色。这种方案避免了原子操作的使用（从而提高了运行时性能），但它可能会在一定程度上增加使用颜色的数量。

## 在不同的GPU设备上运行本算法

为了展示组合算法在三种不同GPU设备上的性能，这三种GPU为NVIDIA K20，NVIDIA K40和NVIDIA P100。由于部分图的顶点和边数过多，占用内存过大，在NVIDIA K20和NVIDIA K40上会出现内存溢出的错误，故只选取了前8个图的运行结果作为对比。NVIDIA K20的配置与前面的实验相同。 NVIDIA K40有2880个CUDA内核和12GB板载内存，NVIDIA P100有3584个CUDA内核和16GB板载内存。图4.5显示了组合算法实现的性能随着GPU功能的增加而很好地提升。

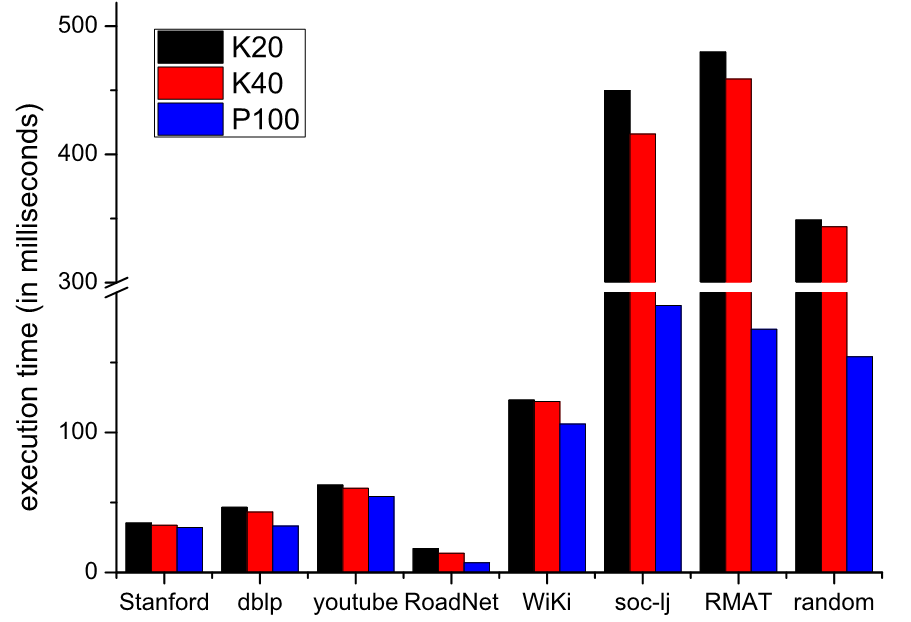


图4.5 在不同GPU上运行本算法

## 本章小结

本章主要针对算法的各个方面进行了测试，除了对于算法运行时间的比较外，为了具体分析算法的最优组合性能及原因，还针对切换时机的选择进行了分析。总体而言，实际对比实验的运行效果与理论分析基本是匹配的，组合算法在所有对比图的运行效率上是优于其他算法的，且是完全着色方案，保证了所有点着色的正确性。

# 总结与展望

随着当前信息时代的日益发展，传统的串行图着色算法在效率上无法满足要求，基于并行架构的大规模图着色的研究也逐渐成为趋势。本课题的目标是设计并实现一种基于并行架构的大规模图着色算法，且该算法能够有效解决在并行过程中的部分难点，并取得效率上的提升。此算法使用CUDA C语言实现，在NVIDIA Tesla K20m GPU上运行通过了正确性测试，并在性能测试的实验中相较于其他算法有了性能上的提升，达到了课题的预期目标。具体而言，本论文所做的研究工作可以总结为以下几点：

1. 分析了GPU具体的硬件架构及对应的在编程中应该注意的特点，总结了在图数据的并行处理中存在的难点，即并行度不够、负载不均衡、长尾问题等，以及难点对应的普遍的解决方案和优化策略，即采用对应的数据结构实现连续访存和开辟足量线程隐藏系统延迟。
2. 优化了一种以顶点为中心的，通过两阶段核函数来避免原子操作的并行图着色算法，解决了反复着色、并行度不足、长尾问题等问题。
3. 优化了以边为中心的类pagerank的算法，将其与本算法进行结合。用本算法每轮的高着色顶点比例收尾，用以边为中心的高速对多数顶点着色作为开始，并依据每种算法特性使用对应的数据结构，且在切换时进行预处理。充分利用两者优点，互补两者的缺点，分析了两个算法最佳的切换时机，使得组合算法在所有测试图上均取得了效率提升。
4. 测试分析了设计的并行算法及优化思路的可行性，针对具体的问题对应的指标进行了实验。
5. 测试了具体性能提升的原因对应的实验数据，即有效指令比例和每轮正确着色顶点比例，以及在普遍使用的图数据上的与其他算法性能对比的数据，结果表明，该算法相较于CPU下的贪心着色算法有1.51-125.81倍的性能提升，相较于JPL（Jones-Plassman-Luby）有8.37-71.07倍的性能提升。

虽然本算法解决了并行编程中存在的部分问题，取得了效率上的提升，基本实现了预期的目标，但还是存在许多不足和需要改进的地方，主要为细节的优化和后续的学习方向。主要有以下几点：

1. 负载不均衡。以顶点为中心的算法存在的固有问题，由于顶点度数存在较大差异，每个线程处理一个顶点，所需时间基本与度数成正比，所以造成负载不均。后续可以针对顶点度数对顶点进行分类，将待处理的顶点放入不同的队列，并用不同的流去处理不同的队列。这要可以利用流的特性将三个队列进行一定程度的并行，但是流的创建，顶点分类也有开销，还需后续实验分析。
2. 切换过程不能自动实施。切换时机需要依据图的特点和具体实验分析才能得出最佳时机，后续需要能够设计一个参数及阈值，以便能够在程序运行过程中自动选择切换时机。
3. 面向动态图对算法进行扩展和优化。
4. 只能在单个GPU上运行。当前工作还只能支持单个GPU，能处理的图数据的大小受制于GPU的最大内存（处理最大图有1亿个顶点和10亿条边）。后续可以针对多GPU系统的特性对本算法进行调整和对应的优化。

致 谢

论文完成之际，首先要感谢我的导师石宣化教授和学长郑志高。他们以沉浸于对应学科多年的经验，凭远高于我的视野，及超乎常人的耐心，从毕设的选题到最终的论文完成均给予了充分的指导。其次要感谢四年的老师给予的专业知识和学习能力的铺垫，以及同学们营造的良好的氛围和环境。最后要感谢一直相伴的亲人朋友自然流淌的关怀和依靠。

感谢对应领域精英们的或晦涩或清晰的论文，感谢分享自己消化总结后的知识和经验的同行们。感谢空无一人的沉寂，感谢喧闹操作的烟火气。感谢操场上挥洒脚步和汗水的夜晚，感谢电脑前跳动手指和思维的午后。感谢深夜明亮的灯，感谢清晨滚烫的粥。感谢思路上误入的歧途，感谢算法里迸发的灵感。

一路坎坎坷坷，满程跌跌撞撞。曾以为的雄关铁道，回头再看，不过咬牙一瞬；曾遇到的艰难困苦，昂首再想，也就释然一笑。世上最好的成语莫不过虚惊一场，天下最可惜的放弃难胜于我本可以。顺利也好，艰涩也罢，全程的经历仿佛是下一步的机关，不试试便难有下一幕的情节。终究是写到了终点，对其中的一切都是感谢与怀念。

最后的最后，要坚信所有的结局都是完美的结局，如果不够完美，说明还没到结局。

参考文献

1. Hang Liu and H. Howie Huang. Enterprise:Breadth-first graph traversal on gpu servers. In International Conference for High Performance Computing,Networking, Storage and Analysis (SC), 2015.
2. M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam. 2016. Parallel Graph Coloring for Manycore Architectures. In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS’ 16). 892–901. https://doi.org/10.1109/IPDPS.2016.54.
3. Dipartimento di Scienze dell’Informazione of the Università degli studi di Milano. 2002. Laboratory for Web Algorithmics. http://law.di.unimi.it/index.php.Accessed July 17, 2018.
4. Gregory J Chaitin. 1986. Register allocation and spilling via graph coloring. US Patent 4,571,678.
5. H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos:Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel and Distrib. Comput. 74, 12 (2014), 3202 –3216.https://doi.org/10.1016/j.jpdc.2014.07.003.
6. A.V. Pascal Grosset, Peihong Zhu, Shusen Liu, Suresh Venkatasubramanian, and Mary Hall. 2011. Evaluating Graph Coloring on GPUs. In Proceedings of the 16thACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPoPP’ 11). 297–298.
7. TIM KALER, WILLIAM HASENPLAUGH, TAO B. SCHARDL, and CHARLES E.LEISERSON. 2016. Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling. ACM Transactions on Parallel Computing 3, 1 (July 2016), 1–31. https://doi.org/10.1145/2896850.
8. Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMDEfcient Graph Processing on GPUs. In Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT ’15). IEEE Computer Society, Washington, DC, USA, 39–50. https://doi.org/10.1109/PACT.2015.15.
9. Jure Leskovec. 2009. Stanford Network Analysis Project. http://snap.stanford.edu/index.html. Accessed July 11, 2018.
10. Rhyd Lewis, Jonathan M. Thompson, Christine L. Mumford, and J. Gillard. 2012.A wide-ranging computational comparison of high-performance graph colouring algorithms. Computers & Operations Research 39, 9 (January 2012), 1933–1950.
11. Jinkun Lin, Shaowei Cai, Chuan Luo, and Kaile Su. 2017. A Reduction based Method for Coloring Very Large Graphs. In Proceedings of the Twenty-Sixth International Joint Conference on Artifcial Intelligence, IJCAI-17. 517–523. https://doi.org/10.24963/ijcai.2017/73.
12. Maxim Naumov, Patrice Castonguay, and Jonathan Cohen. 2015. Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU.Technical Report NVR-2015-001. NVIDIA. 23 pages.
13. S. Sallinen, K. Iwabuchi, S. Poudel, M. Gokhale, M. Ripeanu, and R. Pearce.2016. Graph Colouring as a Challenge Problem for Dynamic Graph Processing on Distributed Systems. In Proceedings of the International Conference on High Performance Computing,Networking,StorageandAnalysis(SC’16).347358.https://doi.org/10.1109/SC.2016.29.
14. Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. Eﬀective and Efcient Dynamic Graph Coloring. Proceedings of the VLDB Endowment 11,3 (Nov. 2017), 338–351. https://doi.org/10.14778/3157794.3157802.
15. Hao Lu, Mahantesh Halappanavar, Daniel G. Chavarría-Miranda, Assefaw Hadish Gebremedhin, Ajay Panyala, Ananth Kalyanaraman: Algorithms for Balanced Graph Colorings with Applications in Parallel Computing. IEEE Trans. Parallel Distrib. Syst. 28(5): 1240-1256 (2017).
16. Hao Lu, Mahantesh Halappanavar, Daniel G. Chavarría-Miranda, Assefaw Hadish Gebremedhin, Ananth Kalyanaraman: Balanced Coloring for Parallel Computing Applications. IPDPS 2015: 7-16.
17. Aijun Dong, Xin Zhang: Equitable coloring and equitable choosability of graphs with small maximum average degree. Discussiones Mathematicae Graph Theory 38(3): 829-839 (2018).
18. Hong C, Chen D, Chen W, et al. MapCG: Writing Parallel Program Portable between CPU and GPU. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, ACM New York, NY, USA, 2010: 217-226.
19. Gonzalez J E, Low Y, Gu H, et al. PowerGraph: Distributed Graph-Parallel Com-putation on Natural Graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, 2012: 17-30.
20. Harris M. GPGPU: General-Purpose Computation on GPUs. In: Proceedings of SIGGRAPH '04 ACM SIGGRAPH 2004 Course Notes, ACM New York, NY, USA, 2004:3.
21. The University of Florida: Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.
22. Sudev Naduvath, K. P. Chithra, S. Satheesh, Johan Kok: On certain parameters of equitable coloring of graphs. Discrete Mathematics Algorithms and Applications 9(4): 1-11 (2017).
23. R.M.R. Lewis. A guide to graph coloring: algorithms and applications. Springer. 2015.
24. M. K. Taş, K. Kaya, and E. Saule. 2017. Greed Is Good: Parallel Algorithms for Bipartite-Graph Partial Coloring on Multicore Architectures. In 2017 46th International Conference on Parallel Processing (ICPP). 503–512. https://doi.org/10.1109/ICPP.2017.59.
25. Alexandre Rok and Bartosz Walczak. 2014. Outerstring Graphs are χ -bounded. In Proceedings of the Thirtieth Annual Symposium on Computational Geometry (SOCG’14). ACM, New York, NY, USA, Article 136, 8 pages. https://doi.org/10.1145/2582112.2582115.
26. Ümit V. Çatalyürek, John Feo, Assefaw H. Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. 2012. Graph coloring algorithms for multi-core and massively multithreaded architectures. Parallel Comput. 38, 10 (2012), 576–594. https://doi.org/10.1016/j.parco.2012.07.001.
27. Dániel Marx. 2004. Graph colouring problems and their applications in scheduling. Periodica Polytechnica Electrical Engineering 48, 1-2 (2004), 11–16.
28. Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL ’14). Association for Computational Linguistics,55–60.
29. Marek Kubale. 2004. Graph Colorings. American Mathematical Society.
30. Kishore Kothapalli and Sriram Pemmaraju. 2011. Distributed Graph Coloring in a Few Rounds. In Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC ’11). ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/1993806.1993812.