中国科学: 信息科学 2023 年 第 53 卷 第 2 期: 211–233
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
评述

图计算在 ATPG 中的应用探究

毛伏兵[1,2,3,4], 彭达[1,2,3,4], 张宇[1,2,3,4*], 廖小飞[1,2,3,4], 姜新宇[1,2,3,4], 杨赟[1,2,3,4], 金海[1,2,3,4], 赵进[1,2,3,4], 刘海坤[1,2,3,4], 王柳峥[5]

1. 华中科技大学大数据技术与系统国家地方联合工程研究中心, 武汉 430074
2. 华中科技大学服务计算与系统教育部重点实验室, 武汉 430074
3. 华中科技大学集群与网格计算湖北省重点实验室, 武汉 430074
4. 华中科技大学计算机科学与技术学院, 武汉 430074
5. 华为海思半导体有限公司, 深圳 518129

* 通信作者. E-mail: zhyu@hust.edu.cn

收稿日期: 2021–08–06; 修回日期: 2021–12–26; 接受日期: 2022–03–21; 网络出版日期: 2023–02–02

国家自然科学基金 (批准号: 61832006, 61825202, 62072193)、中央高校基本科研业务费资助 (HUST)(批准号: 2020kfyXJJS018, 2020kfyXJJS109) 项目

摘要 ATPG (automatic test pattern generation) 是 VLSI (very large scale integration circuits) 电路测试中非常重要的技术, 它的好坏直接影响测试成本与开销. 然而现有的并行 ATPG 方法普遍存在负载不均衡、并行策略单一、存储开销大和数据局部性差等问题. 由于图计算的高并行度和高扩展性等优点, 快速、高效、低存储开销和高可扩展性的图计算系统可能是有效支持 ATPG 的重要工具, 这将对减少测试成本显得尤为重要. 本文将对图计算在组合 ATPG 中的应用进行探究; 介绍图计算模型将 ATPG 算法转化为图算法的方法; 分析现有图计算系统应用于 ATPG 面临的挑战; 提出面向 ATPG 的单机图计算系统, 并从基于传统架构的优化、新兴硬件的加速和基于新兴存储器件的优化几个方面, 对图计算系统支持 ATPG 所面临的挑战和未来研究方向进行了讨论.

关键词 图计算, 超大规模集成电路, 自动测试向量生成, 电子设计自动化, 电路测试

1 引言

电路测试是超大规模集成电路 (very large scale integration circuits, VLSI) 设计中必不可少的步骤 [1~4]. 对于生产出来的芯片, 必须通过测试的方法确定其功能是否能达到设计的预期. 随着电路集成度的增加, 电路中出现缺陷的可能性增加, 测试的难度提升, 总成本中测试所占的比重变大. 因此, 电路测试的重要性越来越凸显出来.

电路测试的原理是在输入产生激励信号, 然后将其产生的输出信号与预期的信号值进行比较, 输入的激励信号称为测试向量. 在过去, 测试向量由人工计算产生. 随着电路规模的扩大, 这种人工

引用格式: 毛伏兵, 彭达, 张宇, 等. 图计算在 ATPG 中的应用探究. 中国科学: 信息科学, 2023, 53: 211–233, doi: 10.1360/SSI-2021-0267
Mao F B, Peng D, Zhang Y, et al. Research on the application of graph processing in ATPG (in Chinese). Sci Sin Inform, 2023, 53: 211–233, doi: 10.1360/SSI-2021-0267

计算的方式失去了可行性. 为了满足 VLSI 的测试需求, 自动测试向量生成 (automatic test pattern generation, ATPG) 技术得到长足发展. ATPG 技术大大减少了测试的时间和人力成本, 它在保证高故障覆盖率的前提下尽量压缩测试向量集的大小, 并且能轻松、无风险地部署到电路设计和测试的流程中. 但是, ATPG 在整个电路设计过程中仍占用很大一部分时间. 甚至 ATPG 的时间/空间开销已经随电路复杂度的提升成为 VLSI 电路设计的瓶颈, 如何使 ATPG 算法高效执行的问题亟待解决.

为了加速 ATPG 过程, 研究人员提出了各种 ATPG 系统, 其中比较典型的有 PP-TGSt [5]、PROPER-TEST3 [6]、HIPERTEST490 [7,8]、CAS 模式并行测试生成系统 [9,10]、GATTOB [11] 等. 上述系统都是通过如下 3 个方面来优化 ATPG: 可测试性设计 (design for testability, DFT)、串行算法优化和算法的并行化. 其中, 随着硬件条件的不断提升, ATPG 算法的并行化越来越成为主要的研究方向. 尽管学术界和工业界都在 ATPG 的并行化上做出了许多努力, 现有的并行 ATPG 方法仍然存在负载不均衡、并行策略单一、存储开销大和数据局部性差等问题 [1~4].

本文针对图计算在组合 ATPG 中的应用进行了探究. 图计算 (graph processing) 是研究事物之间的关联关系并对其进行建模、分析和计算的图处理技术 [12]. 由于图计算的高效并行度和高扩展性等优点, 快速、高效、低存储开销和高可扩展性的图计算系统可能是有效支持 ATPG 的重要工具, 这对减少测试成本来说尤为重要. 事实上, 在电子设计自动化 (electronic design automation, EDA) 的其他领域已经有很多尝试应用图计算的工作 [1~4]. 美国的国防高级研究计划局 (Defense Advanced Research Projects Agency, DARPA) 提出了名为 IDEA (innovation development & entrepreneurship accelerator) 的项目计划, 该项目由耶鲁大学 (Yale University) 负责. 其目的是基于图计算系统实现一个电路布局生成器, 使用者在掌握的 EDA 知识有限的情况下能通过该生成器完成单板计算机等电子硬件的设计 [1~4]. 我们分析了现有图计算系统在实现组合 ATPG 时可能遇到的各种挑战, 并据此提出了一种专门面向组合 ATPG 的单机图计算系统. 最后从传统架构、新兴硬件加速器和新兴存储硬件几个方面对图计算系统支持 ATPG 的未来挑战和研究方向进行了初步探讨.

2 ATPG 背景与挑战

2.1 ATPG 简介

电路测试是判断生产的电路器件是否合格的过程. 具体来说, 测试是给电路输入产生若干组激励信号, 然后观察电路的输出信号是否符合预期. 如果所有的输出符合预期, 则说明生产的硬件与设计过程预期的功能一致; 反之, 则说明硬件产生了故障, 这时需要定位故障的位置. 用于测试的输入激励信号称为测试向量, 测试向量集的不同决定了测试过程的质量, 它主要由两个指标来判定: (1) 测试向量集的大小. 一般来说集合中测试向量的数量越少越好, 因为测试时间和测试向量的数量成正比; (2) 测试向量集的故障覆盖率. 故障覆盖率 (fault coverage, FC) 是电路测试中一个重要概念, 它是测试向量集能够检测的故障数与电路可能产生的所有故障数的比值. 合格的测试向量集要达到 95% 以上的故障覆盖率 [1~4].

如何在保证较高覆盖率的情况下获取尽可能小的测试向量集是电路测试的关键问题. 在最初的工业界, 测试向量集是工程师根据设计经验以人工计算的方式来获取的. 随着大规模电路的出现, 测试向量集的获取难度急剧上升, 相关计算的复杂度已经无法人为处理. 在此情况下, 自动测试向量生成 (ATPG) 技术得到飞速发展 [1~4]. 但是 Fujiwara 和 Toida [13] 的工作已证明, 即使对于单调电路, ATPG 也是 NP 完全问题. 也就是说, 在最坏的情况下, ATPG 算法的时间复杂度与电路的规模呈指

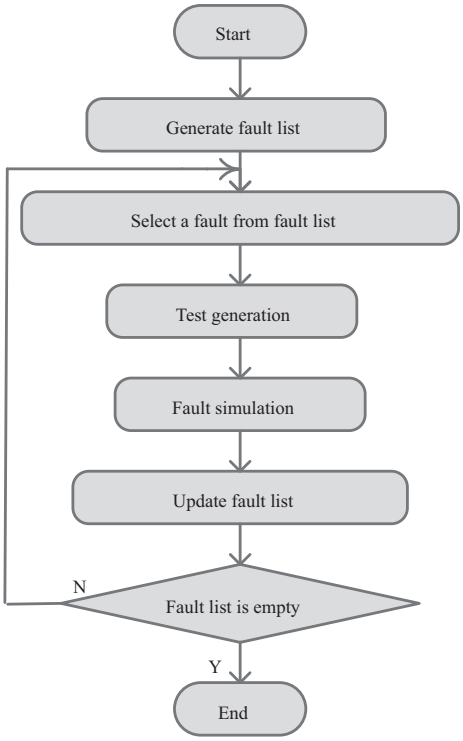

图 1 ATPG 一般流程
Figure 1 General ATPG flow

数相关. 因此, ATPG 问题一直是 EDA 领域的重要研究课题和瓶颈之一.

图 1 给出了 ATPG 的一般流程. 在预处理过程 ATPG 会分析电路中所有可能出现的故障模型, 这些故障模型的集合称为故障列表. 故障模型的目的是将电路的物理缺陷等效于逻辑故障, 它可以划分为不同类型. 在 ATPG 中只考虑单固定型故障模型, 即电路中某一条线路的信号值固定为 0 或 1 [14]. 在生成故障列表后, ATPG 从故障列表中选出一个故障模型进行测试生成和故障模拟, 这是最核心的两个过程. 测试生成为给定的故障模型求解能检测到该故障的测试向量, 它是 ATPG 性能的关键. 对于生成的测试向量, 还需要通过故障模拟来判断它是否能检测到故障列表中的其他故障模型, 故障模拟同样很大程度上影响着 ATPG 的性能. 测试生成选定的故障模型和故障模拟检测到的故障模型被一并从当前故障列表中排除, 然后 ATPG 继续重复两个核心过程, 直到故障列表为空.

2.2 现有 ATPG 加速技术

目前, 大致有下面 3 类方法被用于加速 ATPG: 可测试性设计 (DFT)、串行算法优化和算法的并行化 [1~4].

可测试性设计. 与另外两种加速电路测试过程本身的方法不同, DFT 的理念是在电路的设计阶段就考虑如何组织电路结构使得之后的测试阶段更加简单 [15, 16]. DFT 可被视为一系列技术的集合, 其中有两类比较典型的技术: 扫描方法和内建自测试 (built-in self test, BIST) 方法. 扫描方法采用诸如 LSSD (level-sensitive scan design) [17]、扫描路径 [18] 等移位寄存器技术, 将时序电路的 ATPG 问题简化为组合电路的 ATPG 问题. 对于全扫描设计电路, 只需要针对组合电路开发一种快速有效的 ATPG 算法就足够了. BIST 方法则是在芯片设计时在电路中加入专门的测试模块 [19]. 这类加入了

测试模块的芯片能够通过 BIST 电路进行自测试, 而不需要使用复杂的自动测试设备 (automatic test equipment, ATE).

串行算法优化. 串行算法的优化主要针对测试生成过程, 其类型大致可以分为两大类: 一类是基于拓扑的结构式测试生成算法 [1~4,20]. 这类方法分析电路的拓扑结构信息, 跟踪故障电路中错误信号的传递过程. 在跟踪的过程中, 算法通过调整输入激励使得错误信号能够敏化到输出, 最终得到需要的测试向量. 其中, D 算法 [21,22]、PODEM 算法 [23] 和 FAN 算法 [24] 是最有代表性的结构式 ATPG 算法. D 算法 [21,22] 是已知最早的 ATPG 算法, 它于 1966 年就已发布. 该算法试图通过对电路的所有节点进行结构搜索来确保错误信号的传播, 因此决策树很大. PODEM [23] 和 FAN [24] 对此进行了改进, PODEM 算法将搜索空间限制为电路的主要输入; 而 FAN 算法进一步引入约束线、自由线和线头的概念, 在跟踪过程中仅在线头进行判定, 进一步减小了决策树的大小. 结构式的测试生成算法提出较早, 也得到了广泛的关注和研究, 早已应用于工业界的芯片生产中.

另一类是基于符号的代数式测试生成算法 [25]. 这类方法不依赖于电路的拓扑信息, 而是通过布尔函数的形式对电路进行描述, 然后利用布尔可满足性 (Boolean satisfiability problem, SAT) 或其他方法对函数进行求解以获得所需要的测试向量. Shi 等 [26] 评估了 ATPG 中不同 SAT 求解器的性能, 并挖掘出针对特定问题用启发式算法加速 SAT 的潜力. 此外, 它们通过有效的内存分配减少了生成合取范式 (conjunctive normal form, CNF) 所需的时间. Tafertshofer 等 [20,27] 提出了另一种基于 SAT 的技术, 他们通过在蕴涵图结构上执行故障传播, 加速了对布尔网络的分析. Safarpour 等 [28] 的工作通过将变量标记为非活跃的来加速 SAT. 以与门为例, 如果其输入之一是 0, 那么逻辑锥中的所有变量都可以被 SAT 求解器忽略. 代数式的测试生成方法潜力很大, 但发展不够成熟, 目前只在学术界受到关注, 在工业界中鲜有应用.

并行 ATPG. 随着硬件条件和计算资源的不断丰富, 越来越多研究开始关注于 ATPG 的并行化. 测试生成和故障模拟作为 ATPG 的两个重要过程, 在并行化方面都得到了一定程度的发展. 对于组合电路, 现在大体有两类故障模拟的并行化方案: 并发式故障模拟 [29~32] 和并行模式单故障传播 [33~38]. 并发式故障模拟同时对多个电路进行模拟, 其中包括一个无故障电路和多个故障电路. 并发式故障模拟的最大好处在于它灵活和适应性强, 从而使得它在工业界被广泛采用. 早期工作中 [30~32] 提出了启发式方法用来提高并发式故障模拟的速度. 然而, 并发式故障模拟需要维护多个电路的数据, 因此存在存储开销大的问题. 单故障传播方法则预先计算好无故障电路的模拟结果, 之后每次只处理一个故障, 然后将模拟结果与预存的无故障电路比对. 单故障传播方法与并发式故障模拟相对比, 存储开销要小很多, 而且运行时只需要模拟和故障相关的子电路, 所以时间开销也相对较小. 对于组合电路, 采用测试向量并行可以提高单故障传播方法的性能, 此方法为并行模式单故障传播 (parallel pattern single fault propagation, PPSFP) [34~36]. PPSFP 能够实现与并行处理的测试向量数相同的加速比, 而且还能和隐式的故障模拟算法结合来进一步提升性能 [33,35,39]. 临界路径追踪算法是 Antreich 和 Schulz [34] 设计的一种隐式故障模拟算法, 该算法能快速排除无扇出区域的可检测故障. Harel 等 [39] 在单故障传播中使用支配点的概念, 利用支配点能更方便找出电路中的无扇出区域. Maamari 和 Rajski [35] 则提出了茎区域概念. 上述 3 种算法都能很方便地和 PPSFP 结合在一起, 以减少电路中的显式故障模拟区域. 测试生成的并行化策略也有一些研究工作, 表 1 给出了测试生成的几种典型并行化策略 [29~39].

同时, 有大量研究用 GPU 加速 ATPG 方法 [40~43]. 现有的基于 GPU 的测试生成的研究主要通过比特位级别并行来提高并行性 [43]. 现有的故障模拟主要通过算法并行 (并行化故障模拟算法)、模型并行 (把电路划分为不相交的子部分)、数据并行 (向量并行和故障并行) 来提高故障模拟的并行度 [40,43]. 目前电路规模逐渐增大, 由于 GPU 内存有限, 在支持 ATPG 算法时, 现有方法存在主机和 GPU 之间

表 1 并行测试生成方法
Table 1 Parallel approaches of test generation

Parallel strategies	Introduction
Fault-level parallelism	Fault-level parallelism divides the fault list into multiple sub-fault lists, and then each distributed node processes the corresponding sub-fault lists in parallel. The upside is that coding is easy, and the communication overhead is very low. If the fault list is properly partitioned, this kind of approach can achieve a nearly linear speedup. The downside is the imbalance-load problem. In addition, the size of its test vector set may increase compared to the serial method.
Heuristic parallelism	The purpose of heuristic parallelism is mainly to improve the final fault coverage. Structured serial test generation algorithms often adopt different heuristic rules. However, the various heuristic rules used to guide test generation are not universal, and their effects will have their own advantages and disadvantages according to the different faults handled. Heuristic parallelism means that each distributed node utilizes different test generation algorithms to deal with the same fault. When a node generates a corresponding test vector, it will immediately notify other nodes to terminate the current work and start processing the next fault at the same time. This kind of method can guarantee a higher coverage than the serial algorithm, but the reduced time overhead is very limited.
Search space parallelism	Search space parallelism divides the decision tree into multiple independent sub-search spaces, each of which is statically or dynamically assigned to node processing. Its upsides are low communication overhead and high parallelism, but its coding is relatively difficult.
Function-level parallelism	Function-level parallelism is to divide the serial algorithm into unrelated subtasks, and then assign the subtasks to different nodes for parallel processing. This approach must ensure that there are no common variables between subtasks. Otherwise, it will greatly increase the difficulty of parallel control, and in the worst case, it can only execute the subtasks in a serial order. At the same time, because each subtask performs completely different processing, it is difficult for developers to estimate the workload of different tasks for load balancing related processing.
Circuit parallelism	Circuit parallelism is adopted when the circuit size is so large that the memory of a single node cannot hold the relevant data of the entire circuit. It divides the topology of the circuit into sub-topologies and stores them in different nodes, and its performance greatly varies depending on the division method. Existing partitioning methods often lead to a huge communication overhead, and it is difficult to guarantee the parallelism of the algorithm.

的数据传输开销大、GPU 内存消耗大、GPU 有效利用率低等问题, 导致可扩展性差 [40~43]. 然而, 本文图计算方法可以通过挖掘电路图的拓扑信息, 动态识别每一轮迭代中需要处理的电路图数据以及精确评估任务负载, 通过只加载需要处理的电路图数据到 GPU 中, 并基于识别的负载进行任务分配, 从而提高 GPU 的利用率以及保证 GPU 的负载均衡. 同时, 也可以通过分析电路图的拓扑信息挖掘各并行任务数据访问之间的空间/时间关联性, 规则化各任务的数据访问行为, 使得大量任务能够共享电路图数据的存储和访问, 极大减少电路图的冗余存储和访问开销, 从而减少 GPU 的内存消耗以及主机和 GPU 之间的数据传输量. 这使得我们方法具有很好的可扩展性, 未来能够高效支持大规模电路图分析.

2.3 并行 ATPG 的挑战

无论是 DFT 技术还是对于 ATPG 串行算法的优化, 都已经发展到相当成熟的地步. 近年来, 有关这方面的新成果已经非常少, 而且基本都是有关代数式的测试生成算法的工作, 在工业界还没有得到

应用. 随着硬件水平的提升, ATPG 的并行化已经成为加速电路测试的最有潜力的研究方向之一. 但是, 现有并行 ATPG 的相关工作还面临着诸多挑战.

(1) **负载均衡问题.** 现有的并行 ATPG 方法对子任务的划分还停留在较为粗粒度的层次. 以测试生成的故障级并行为例, 该类方法将故障列表划分为子列表. 多个处理器对这些子列表进行并行处理, 也就是说处理器之间对不同的故障进行测试生成. 而不同故障的测试生成过程的时间开销差距是十分大的, 而且这个开销很难预估. 这可能导致工作负载的严重不平衡, 使并行给 ATPG 带来的收益大打折扣.

(2) **并行策略单一.** 无论是故障模拟还是测试生成过程都存在多种并行性, 但是现有的并行 ATPG 系统中往往只采用某一种并行策略. 不同的并行策略各有优劣, 甚至在并行性上不互斥, 对这些并行策略进行有机组合相较于执行单一的并行策略在性能上有更大的提升空间. 另外, 在现有的并行 ATPG 系统中, 测试生成和故障模拟过程是紧耦合的. 也就是说, 每个并行任务既进行测试生成, 也进行故障模拟. 然而, 测试生成和故障模拟之间的并行性并非完全一致. 因此耦合方式也限制了并行 ATPG 系统能够采用的并行策略.

(3) **存储开销大, 数据局部性差.** 传统的 ATPG 系统以链表的形式表达电路的拓扑, 而将相关的所有状态数据全部放在同一个结构体中. 这种存储方式使得并行任务必须维护单独的电路副本, 在电路规模越来越大的情况下产生大量的空间开销. 除此之外, 大部分 ATPG 算法往往只处理少数几类状态数据. 结构体链表的数据结构使得同种状态数据的存储在物理上是十分离散的, 最终导致计算时访存效率低下.

3 图计算用于 ATPG

将图计算应用于 ATPG 中是一种可行的并行化方案. 电路的拓扑结构可以抽象为一般图, 而故障模拟和测试生成都有很多结构式的算法是在电路拓扑的基础上执行的, 这些算法可以转换为图算法. 图计算中对并行任务的划分是细粒度的, 它会把不同的顶点分配给不同线程进行处理. 这在 ATPG 中相当于将单个逻辑门的处理视为一个可并发的子任务, 而传统 ATPG 方法对逻辑门的处理都是串行的. 通过这种逻辑门级别细粒度的划分, 能够进一步提高 ATPG 的并行度, 并且也更便于进行负载均衡. 除此之外, 电路在图计算系统中的存储方式也将发生很大变化. 传统的 ATPG 系统以链表的形式表达电路的拓扑, 而相关的所有状态数据全部放在同一个结构体中. 这种存储方式不仅会产生较大空间开销, 并且数据局部性也很差. 图计算则完全将图结构数据和状态数据分离开来: 图结构数据可以用压缩稀疏行 (compressed sparse row, CSR) 等压缩格式 [44] 存储, 这样能减少空间开销; 不同类型的状态数据用单独的数组存储, 这样能提高数据的局部性. 我们将在本节中简单介绍基于拓扑结构的组合电路算法在图计算模型下的实现.

3.1 组合电路的图抽象

给定一个组合逻辑电路 C, 如图 2(a) 所示, 电路由输入端集合 PI (primary input)、输出端集合 PO (primary output)、逻辑门集合 CG 和它们之间的连线构成. 输入端、输出端和逻辑门都有信号值 (或称为逻辑值). 本文涉及的逻辑门类型包括: 与门 (AND)、或门 (OR)、非门 (NOT)、异或门 (XOR)、与非门 (NAND)、或非门 (NOR)、异或非门 (XNOR), 以及自定义的扇出结点 (STEM) 等. 图 2(b) 给出了该电路拓扑结构的图抽象, 其中 PI、PO 和逻辑门被抽象为电路中的顶点, 顶点自带类型值. 图中的边用于表示电路中的连线, 因为连线实际上用于信号值的传输, 自带有方向这一属性, 所以图中的边

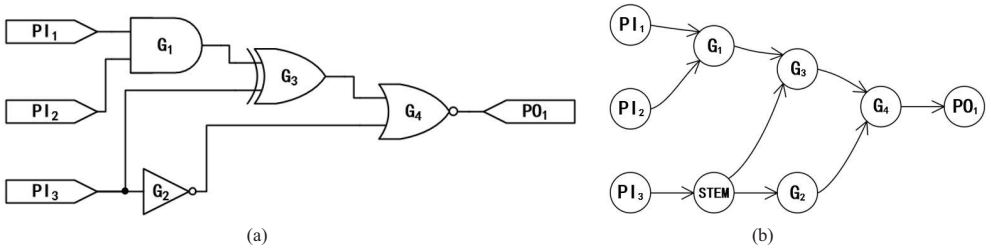

图 2 (a) 电路图; (b) 相对应的带有一个 STEM 类型结点的抽象图
Figure 2 (a) Circuit graph; (b) corresponding abstract graph with a stem node

均为有向边, 方向同信号值传输方向一致. 如果在电路中有元件向多个其他元件输出信号, 即存在一个扇出节点, 那么该节点会抽象为 STEM 类型的顶点 [4].

3.2 ATPG 算法转化为图算法介绍

我们用图计算中常用的 Gather-Apply-Scatter (GAS) 模型来说明如何将测试算法以图算法的形式描述 [3, 44]. GAS 模型采用以顶点为中心的编程模型, 将图处理中的一轮迭代分为 3 个阶段: Gather, Apply 和 Scatter. 在 Gather 阶段, 活跃顶点 u 收集邻接顶点和边的状态值, 然后对这些状态值求广义和, 即

$$\sum \leftarrow \bigoplus_{v \in \mathrm{Nbr}[u]} g(D_u, D_{(u,v)}, D_v), \tag{1}$$

其中, D_u, D_v 和 $D_{(u,v)}$ 分别代表顶点 u、顶点 v 和边 (u,v) 的状态值, 顶点 v 是顶点 u 的邻接顶点. 广义和运算 $\bigoplus$ 是由用户自定义的操作, 它必须满足交换律和结合律. 式子中求出的值 $\sum$ 被 Apply 阶段用于更新活跃顶点 u 的状态值:

$$D_u^{\mathrm{new}} \leftarrow a\left(D_u, \sum\right). \tag{2}$$

最后, Scatter 阶段用更新后的活跃顶点状态值 D_u^{new} 来将活跃顶点邻接边的状态值进行更新:

$$\forall v \in \mathrm{Nbr}[u] : (D_{(u,v)}^{\mathrm{new}}) \leftarrow s(D_u^{\mathrm{new}}, D_{(u,v)}, D_v). \tag{3}$$

在 Scatter 阶段还会生成新的活跃顶点集合, 用于下一轮迭代. 当活跃顶点集为空时, 图算法达成收敛. 对于有向图, 在 Gather 和 Scatter 阶段的扇入和扇出根据具体的算法确定. 比如, 在 PageRank 中, Gather 阶段只对入边操作, Scatter 只对出边操作.

在 GAS 模型下, 必须由用户定义 4 个接口函数: Gather(), Sum(), Apply() 和 Scatter(). 在 Gather 阶段, Gather() 和 Sum() 分别用于收集邻接顶点的状态和对所有状态求广义和. Gather() 并行处理邻接表中的边, 将边和顶点的数据传给 Sum() 做广义和计算. Gather() 可调用的边集可以是空集、入边、出边和所有邻接边, 具体由用户来决定. Gather 阶段完成后, Apply() 获取最后的累加值, 并根据累加值计算新的顶点状态. 计算出来的顶点状态通过原子操作写回到对应的活跃顶点. 在 Scatter 阶段, Scatter() 也并行处理邻接表中的边, 根据边和更新后的顶点的状态产生新的边值 $D_{(u,v)}^{\mathrm{new}}$. 对于很多图算法, 在 GAS 模型下并不需要维护邻接边的状态值, 这时 Scatter() 只用于活跃顶点集的更新.

逻辑仿真是电路测试过程较为简单的算法, 它的目的是根据给定的 PI 的逻辑值计算出电路 PO 的逻辑值. 算法 1 给出了逻辑仿真在 GAS 模型下的图算法实现, 它实现了 GAS 模型的 4 个接口函数, 最后由 LogicSim() 调用这些函数以实现逻辑仿真的功能. 具体来说, Gather() 收集的是邻接顶点

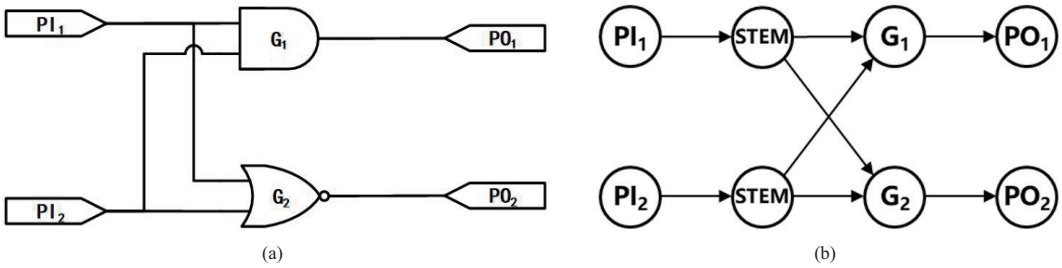

图 3 (a) 电路图; (b) 相应的带有多个 STEM 类型结点的抽象图
Figure 3 (a) Circuit graph; (b) corresponding abstract graph with multiple stem nodes

的逻辑值, 并在 Sum() 根据顶点类型对收集的逻辑值进行逻辑运算. Apply() 则会根据顶点类型判断是否对 Sum() 计算出的逻辑值进行取反, 因为在 Sum() 中 NAND, NOR 和 XNOR 这几个逻辑门是和 AND, OR 和 XOR 做相同的处理. 因为逻辑值是作为顶点的状态值进行存储的, 逻辑仿真只需要维护这唯一的顶点状态值. 因此, Scatter() 只需要更新活跃顶点集, 它将所有前驱顶点都完成逻辑值运算的顶点加入到下一轮迭代的活跃顶点集中.

Algorithm 1 Logic simulation

```
Input: G, Logic.
Output: Logic.
 1: function Gather(u, e, v);
 2:     return v;
 3: end function
 4:
 5: function Sum(a, b, type)
 6:     if type ∈{AND, NAND} then
 7:         return a ∧ b;
 8:     else if type ∈{OR, NOR} then
 9:         return a ∨ b;
10:     else if type ∈{XOR, XNOR} then
11:         return a ⊕ b;
12:     else if type == NOT then
13:         return ¬b;
14:     else
15:         return b;
16:     end if
17: end function
18:
19: function Apply(u, Accum, isActive)
20:     isActive ⇐ isActive − {u};
21:     if u.type ∈{NAND, NOR, XNOR} then
22:         Accum ⇐ ¬Accum;
23:     end if
24:     return Accum;
25: end function
26:
27: function Scatter(u, e, v)
28:     v.count ⇐ v.count + 1;
29:     if D_v.count == D_v.indegree then
30:         isActive ⇐ isActive ∪ {v};
31:     end if
32: end function
33:
34: function LogicSim(G, Logic, Type)
35:     isActive ⇐ ∅;
36:     do
37:         for each vertex ∈ isActive do
38:             for each edge ∈ vertex.inNbr do
39:                 Logic[vertex] ⇐ Sum(Logic[vertex],
    Logic[Gather(vertex, edge, edge.src)], vertex.type);
40:             end for
41:             Logic[vertex] ⇐ Apply(vertex,
                Logic[vertex], isActive);
42:             for each edge ∈ vertex.outNbr do
43:                 Scatter(vertex, edge, edge.dst);
44:             end for
45:         end for
46:     while isActive ≠ ∅;
47:     return Logic;
48: end function
```

以图 3 来举例说明图计算用于逻辑仿真. 初始化 PI_1, PI_2 的逻辑值为 0, 1, 并把两个 PI 设置为活跃顶点. 第 1 轮迭代开始. 因为 PI 没有入边, 因此这一轮迭代跳过 Gather 和 Apply 操作, 通过 Scatter 操作将两个 PI 顶点的逻辑值传递给后续顶点 STEM, 该过程各顶点并行. 两个 STEM 顶点在该轮迭代都收到一次 Scatter 信号, 收到信号次数和 STEM 顶点入度相同, 因此将它们设为第 2 轮迭代的活跃顶点. 第 2 轮迭代开始, 活跃顶点为两个 STEM. 第 1 个 STEM 进行 Gather 操作, 收到一个逻辑值 0, 然后进入 Apply 阶段, STEM 根据自身类型将逻辑值置为 0, 然后通过 Sactter 操作将逻辑值传递给后续顶点 G_1 和 G_2. 第 2 个 STEM 进行 Gather 操作, 收到一个逻辑值 1, 然后进入 Apply 阶段,

STEM 根据自身类型将逻辑值置为 1, 然后通过 Sactter 操作将逻辑值传递给后续顶点 G_1 和 G_2. 顶点在第 2 轮迭代都受到两次 Scatter 信号, 收到信号次数和它们的入度相同, 因此它们是第 3 轮迭代的活跃顶点. 注意, 对两个 STEM 的处理是并行的. 第 3 轮迭代开始, 活跃顶点为 G_1 和 G_2. G_1 进行 Gather 操作, 收到两个逻辑值 0, 1, 然后进入 Apply 阶段, G_1 根据其自身类型 AND, 对逻辑值进行逻辑与操作得到 G_1 逻辑值 0, 然后通过 Sactter 操作将逻辑值传递给后续顶点 PO1. G_2 进行 Gather 操作, 收到两个逻辑值 0, 1, 进入 Apply 阶段, G_2 根据自身类型 NOR 将两个逻辑值进行或非得到逻辑值 0, 然后在 Scatter 阶段将逻辑值传递给 PO_2, 对 G_1 和 G_2 的处理也是并行的. 第 4 轮迭代开始, 活跃顶点为 PO_1 和 PO_2. PO_1 进行 Gather 操作, 收到一个逻辑值 0, 进入 Apply 阶段, PO_1 根据自身类型将逻辑值置为 0, 因为 PO_1 没有出边, 跳过 Scatter 阶段. PO_2 进行 Gather 操作, 收到一个逻辑值 0, 进入 Apply 阶段, PO_2 根据自身类型将逻辑值置为 0, 因为 PO_2 没有出边, 跳过 Scatter 阶段. 最后没有活跃顶点, logic simulation 算法结束.

我们基于学术上流行的开源 ATPG 工具 Atalanta [45] 进行研究, 对 FAN 算法 (通过对电路中的关键节点进行决策) 进行了支持. ATPG 主要包括测试生成和故障模拟, 对于显式故障模拟, 类似于逻辑仿真, 可用上面定义的 4 个接口函数进行实现. 对于测试生成, 在 FAN 算法中的蕴含 (implication)、线确认 (line justification) 和回溯 (backtrace) 3 个阶段用我们定义的 4 个接口函数进行逻辑值的计算.

4 图计算系统相关研究

图计算是以图作为数据模型来表达问题, 并对问题进行解决的过程 [12,44,46~54]. 目前已经出现了单机图计算系统、分布式图计算系统, 以及图计算硬件加速技术. 单机图计算系统能处理小规模电路. 然而, 当电路规模大时 (例如对于 AI 芯片, 图顶点规模达到万亿规模), 单机系统处理慢或是处理不了. 需要用硬件加速技术进行加速或是用分布式系统进行处理. 本节将对目前图计算系统相关研究进行基本介绍.

4.1 单机图计算系统

单机图计算系统能充分利用单机处理图计算任务的能力, 避免了分布式系统下代价高的网络通信开销. 但是这类系统由于硬件资源有限, 使其无法实现好的扩展性, 需要的处理时间通常和图数据规模成比例 [55,56]. 单机图计算系统可分为两类, 面向高端多核、大内存服务器的内存 (in-memory) 图计算系统 [56~60] 和面向商用 PC 的核外 (out-of-core) 图计算系统 [61~67]. 第 1 类在图处理过程中会将图数据完全放入内存中, 第 2 类通常利用磁盘来存放图数据, 它采取一定的划分策略来进行分块处理. 典型的单机图计算系统通常会将图顶点和边划分为小的分片, 以便分片能够在高速的存储设备中进行高效处理, 并提供了充分的并行性和优化的访存模式. 对于内存图计算系统, 图数据进行划分的目的是使每个核心能有效处理各自分配的分片, 有些内存图计算系统不需要显式地运用划分策略, 只需要在计算过程中对任务进行适应性的调度, 就可以取得很好的并行效果. 对于核外系统, 由于图数据无法被全部加载进内存, 图划分策略使得分片能够被加载进内存进行处理. 对于编程模型是以点为中心的图计算系统, 一般将图顶点和边的分片同时加载进内存中. 对于编程模型是以边为中心的系统只将图顶点分片整个加载进内存, 边数据从磁盘上以流的形式进行加载处理. 下面根据内存图计算系统和核外图计算系统两个类别, 对几个典型的单机图计算系统进行基本介绍.

4.1.1 内存图计算系统

单机内存图计算系统是指在单机上运行, 并且图可以直接被完全加载到内存中进行计算. 但是单机的计算能力和内存空间总是有限, 因此只能解决较小规模的图计算问题.

Ligra [56] 是一个基于共享内存的轻量级单机图计算系统. 它的核心思想是根据执行过程中活跃点子集的大小和出度等情况, 自动在 push 计算模式和 pull 计算模式之间进行切换, 从而加快图计算中图遍历算法的收敛. Ligra 适用于共享内存设备, 目前的商用服务器已经可以在内存中处理百亿条边级别的图数据. 对比分布式集群, 单服务器上能实现更优的单核性能并且可靠性更高. Galois [57] 单机图计算系统的核心思想是在数据驱动 (data-driven) 的计算模式下, 充分发挥出自主调度 (autonomous scheduling) 的优势. 它设计了一个机器拓扑感知的任务调度器和一个优先级任务调度器. 同时它提供了相应的扩展库和运行时系统, 并支持多种编程接口. GraphMat [58] 首次提出多核 CPU 最优化顶点编程模型, 对多核可扩展性进行了支持, 并且为用户和开发者提供了友好的编程接口. GraphMat 通过顶点程序以及映射顶点程序到后端的高性能稀疏矩阵算子来运行. Polymer [59] 是一个在多核上考虑非一致内存访问 (non-uniform memory access, NUMA) 特性的图分析框架. 它利用了 NUMA 特性和图的数据分布对图进行分析, 并且提高了全局同步的效率, 负载均衡和数据结构的灵活性. 核心思想是 Polymer 首先根据访问模式对图系统的拓扑数据、应用程序定义数据和可变运行时状态进行差异化分配和放置, 以最大限度地减少远程访问. 此外, 对于一些剩余的随机访问, Polymer 通过使用跨 NUMA 节点的轻量级顶点复制, 将随机远程访问转换为顺序远程访问. 为了改善负载平衡和顶点收敛, Polymer 进一步构建了分层屏障以提高并行性和局部性, 用于倾斜图的面向边的平衡分区, 以及根据活动顶点的比例自适应数据结构.

上述系统中 Ligra 根据图形的稠密情况自适应地在 pull/push 计算模式下进行切换, 并提供基于边映射, 顶点映射以及顶点集映射的并行编程算法. Galois 采用专用语言 (domain-specific languages, DSLs) 写出算法进行图分析工作并在现有 3 种图 DSLs 基础上提供了轻量级的 API, 简化图算法实现. GraphMat 是首次对多核 CPU 进行优化并且以顶点为编程中心的轻量级图计算框架. Polymer 则针对在 NUMA 特性的计算机结构上运行图算法时进行优化.

4.1.2 核外图计算系统

为处理大规模图, 单机核外图计算系统被提出. 单机核外图计算系统是图计算系统运行在单机环境, 并通过计算将图数据通过不断地与内存和磁盘进行交互进行高效的图算法. 它具有较少的硬件开销, 避免了分布式系统中负载不均衡、严重的通信开销、调试和优化难等问题. 但单机核外图计算系统受限于单机计算能力和核外存储系统的数据交换带宽限制. 下述核外图系统, 主要在不同编程模型, 如以点为中心、以边为中心, 以及以路径为中心下对大图进行处理 [61~67].

GraphChi [61] 是编程模型为以点为中心的单机图计算系统. 它提出了并行滑动窗口 (parallel sliding windows, PSW) 技术来优化图计算过程中对磁盘的访问, 同时它使用了空间高效的图数据结构和划分策略. 它具有动态选择性调度模块, 可以在更新或图结构发生改变时, 按需求将某些点进行调度来进一步提高性能. GraphChi 是第 1 个基于磁盘的图计算系统. GraphChi 将边数据进行预处理, 按照特定的格式进行存放, 并用并行滑动窗口的方式减少对磁盘的随机访问次数. 在 GraphChi 系统中, 在一个区间的所有边需要在计算前装载到内存中导致不必要的磁盘数据传输并且在图查询时性能比较差. 为解决这些问题, TurboGraph [62] 采用了名为 'pin-and-slide' 的并行执行模型. TurboGraph 是第 1 个真正的并行图引擎, 它利用了 (1) 包括多核并行和 FlashSSD IO 并行在内的全并行和 (2) CPU 处

理和 I/O 处理得尽可能完全重叠. TurboGraph 还提供引擎级算子, 例如在 pin-and-slide 模型下实现的 BFS (breath first search). VENUS [63] 提出了以顶点为中心的流图处理模型, 显著减少了基于磁盘的图 I/O, 大大改善了系统 I/O 瓶颈的情况. VENUS 旨在实现数据横向扩展和大集群上并发读取大数据, 以应对海量数据所需的大量 I/O. VENUS 拥有基于无关算法的独特技术方法. X-Stream [64] 单机图计算系统是以边为中心的计算思想, 用流的方式处理外存或者内存中的边, 从而提高访存的连续性, 并充分利用存储设备的带宽来提高性能. 它缺乏对选择性调度进行有力支持.

PathGraph [65] 单机图计算系统是以路径为中心 (path-centric) 的图计算方法. 它设计了以路径为中心的压缩存储结构来提高计算过程中的局部性以及数据访问的连续性, 从而加速图计算任务. GridGraph [66] 是在单机上处理大规模图数据的系统. 它对 X-stream 系统的思想进行了改进. GridGraph 对图数据采用了两层划分策略. 它将图顶点进行一维划分, 将图的边划分为二维网格, 同时在运行时进一步对边进行粗粒度的动态划分来提高访存效率. 在执行模式上, 它采用了新颖的双滑动窗口技术, 并且采用灵活的选择性调度策略进行优化. 该系统利用外存进行图计算, 可以达到同分布式系统相当的性能表现. NXgraph [67] 是一种单机上的高效图计算系统. 它通过对顶点间隔和边子分片的抽象, 提出了目标排序子分片结构来存储图. 通过将顶点和边划分为区间和子分片, NXgraph 确保图数据访问局部性并实现细粒度调度. 通过根据目标顶点对每个子分片内的边进行排序, NXgraph 减少了不同线程之间的写入冲突并实现了高度的并行性. NXgraph 提出了 3 种更新策略, 即单相更新 (single-phase update, SPU)、双相更新 (double-phase update, DPU) 和混合相更新 (mixed-phase update, MPU). NXgraph 可以根据图的大小和可用的内存资源, 自适应地为不同的图问题选择最快的策略, 充分利用内存空间, 减少数据传输量. 所有这 3 种策略都利用了简化的磁盘访问模式. 当前的异步方法在跨不同图分区传播状态方面仍然非常不理想. 这为跨分区状态更新带来了瓶颈, 并减慢了处理任务的收敛速度. HotGraph [60] 解决了这个问题, 通过提取主干结构 (称为热图) 来加快图处理, 该主干结构跨越原始图的所有分区. 通过这种方法, 传统解决方案中的大多数跨分区状态传播现在只发生在少数热图分区内, 从而消除了跨分区瓶颈. 同时文中还开发了一种分区调度算法, 通过将热图保存在内存中并尽可能为其分配最高优先级来最大化热图的有效性. 然后提出向前和向后扫描执行策略以进一步加速收敛. HotGraph 提出了异步加速的新方法来处理跨分区中传播状态不理想的瓶颈.

4.2 分布式图计算系统

在大数据时代, 图已经增长到无法放入一台机器的内存中. 即使可以, 性能也会受到内核数量的限制. 单机处理并不是真正可扩展的解决方案. 为了有效地处理大规模图, 分布式图计算系统被提出. 分布式计算机系统也存在一些问题. 分布式图计算需要对图进行分割, 而现实世界中的图复杂的结构加剧了图分割的难度, 很难找到一个高时效并且能保证得到高质量的图分割方法. 当图算法在每次迭代中仅需要更新少数点的值时, 无法充分发挥分布式平台计算资源多的优势, 反而会因网络消息延迟而影响图算法的性能. 分布式图计算系统有负载不均衡、成本高的通信开销以及弱健壮性等缺点 [51]. 下述典型系统主要以同步和异步方式来对大规模图进行分布式处理, Kineograph 系统 [68] 和 Chronos 系统 [69] 支持动态图中的解析计算, 其他典型系统主要支持静态图.

Pregel [70] 是最早用于图处理的分布式系统之一. 它提供了支持大规模图的计算模型, 其图算法通常被表示为一系列迭代, 为获得更高的计算数据访问率. 此系统中的方法可以进行细粒度的并行. Blogel [71] 是 Pregel 系统的拓展, 它解决了 Pregel 在处理现实世界中图的 3 个特征, 倾斜度分布、高密度、大直径时存在性能差的问题. Blogel 是以块为中心的框架并且提供并行算法来有效地将任意图划分为块, 在这些块上运行以块为中心的程序. 与 Pregel 的同步数据推送的 BSP (bulk synchronous

parallel) 模型不同, Distributed GraphLab [72] 支持以异步方式执行图算法. 这种异步模型往往比同步模型具有更快的收敛速度和更低的同步成本. 然而, 它仍然受到顶点度倾斜引起的负载不平衡的影响. PowerGraph [73] 基于 GraphLab, 将整个图的边均匀地划分为几个分区, 并提出在边上分解顶点程序以平衡负载. 作为进一步的优化, PowerSwitch [74] 提出了一种混合执行模式, 允许在同步和异步模式之间动态切换以获得最佳性能. Kineograph [68] 是一个分布式系统, 它采用传入数据流来构建一个不断变化的图, 该图捕获数据馈送中存在的关系. Kineograph 进一步支持图挖掘算法, 采用增量图计算引擎来进行图分析, 以跟上图的持续更新. Chronos [69] 作为用于进化图分析的最先进的特定框架, 使用位置感知批量调度方案来最大化数据局部性的好处. 然而, 它们不知道分区之间的依赖关系, 仍然需要处理几个局部收敛的分区.

4.3 图计算硬件加速技术

加速器因其具备丰富的带宽资源、高的并行能力, 以及低的数据传输延迟等技术优势, 从而使它成为实现高效图计算的重要技术手段之一. 按照加速器物理器件性质来分, 面向图计算的硬件加速方案大体可分为基于 GPU 的图计算加速器、基于 MIC (many integrated core) 的图计算加速器、基于 FPGA (field programmable gate array) 的图计算加速器和基于 ASIC (application specific integrated circuit) 的图计算加速器等 [55].

与 CPU 相比, GPU 因其集成众多计算单元, 可提供更强的并行计算能力, 可以更高效地支撑大规模图顶点遍历和更新. 为了解决基于 GPU 图计算负载不均衡的问题, Back40Computing [75] 实现了基于 GPU 的高性能遍历算法. Gunrock [76] 进一步设计了基于 GPU 的图计算通用加速库. 为进一步提升 GPU 的带宽利用率, Graphie [77] 对大规模图数据的处理调度进行了优化. 由于图应用的数据局部性差以及访存随机性大的特点, 因此, 其访存延迟通常明显大于传统应用. 针对图计算访存延迟高的问题, 现有基于 FPGA 的图计算研究在执行模型和数据划分两方面开展了工作. 在执行模型方面, 已有的 FPGA 工作多数基于边为中心 (edge-centric) 的模型 [78] 以提升边数据访问的局部性 [70, 78]. 在数据划分方面, 传统的划分方法以等分方式对所有边数据以同等计算量为尺度进行划分. 这种方法加剧了各划分内点数据访问的随机性. 通过采用网格划分 (grid-partition) 的方法可以提升点数据访问的局部性 [79]. MOSAIC [80] 是基于 CPU 和 Intel MIC [81] 异构平台实现的核外图计算框架, 其采用的核心思想是将图数据划分成固定大小的 tile, 然后每个 tile 内部的数据在 MIC 协处理器端进行局部加速归约, 对归约得到的结果在 CPU 端再进行全局归约. 由于 tile 本身是一种边表结构, 因而它可以较好地利用带宽资源.

传统 CPU 架构在处理图应用时, 面临访存延迟较高、并行度较低等问题. 现有的图计算系统能在一定程度上缓解上述问题, 但是其性能与性能功耗比同样受限于顶层的硬件架构. 因而, 研究人员尝试为图计算设计专用加速芯片 [55]. 在缓存设计方面, Graphicionado [82] 使用高速暂存存储器 (scratchpad memory) 替代传统的 cache 架构. 通过将点数据访问转移至片上缓存, Graphicionado 显著提升了顶点数据的访存精度, 降低了访存延迟. 在片上缓存架构方面, 比尔肯大学 (Bilkent University) 从细分化角度对传统 cache 架构进行了改进 [83]. 此架构对图应用中所有的数据进行细粒度的划分, 将传统的一致 cache (uniform cache) 细分为多个子 cache, 用以保存不同类型的图数据. 存内计算 (processing in memory, PIM) 可以通过将计算逻辑放置在内存芯片内来减少内存和计算之间的数据移动, 从而减少延时, 提高性能. 基于最新先进技术 (如混合内存立方体, hybrid memory cube, HMC) 的存内计算 (PIM) 架构展示了图计算的巨大潜力. GraphQ [84] 是一种改进的基于 PIM 的图计算架构, 它从根本上消除了不规则的数据移动. DepGraph [44] 提出了一种依赖驱动的可编程加速器. DepGraph 提出了一种有

效地依赖驱动异步执行方法, 用于新型微架构设计, 以实现更快的状态传播.

5 图计算应用于组合电路 ATPG 的挑战

图计算应用于组合电路 ATPG 的挑战主要包括 ATPG 算法转化为图算法的挑战、ATPG 对应的图算法高效执行面临的挑战. ATPG 算法转化为图算法的挑战主要有两点: (1) 大多数 ATPG 算法, 例如 D 算法、PODEM 算法和 FAN 算法, 都是用邻接链表存储电路图数据的, 存在存储和处理开销大的问题. (2) 现有的图计算系统不支持 ATPG 算法. 例如传统的 ATPG 算法 (如学术上流行的开源工具 Atalanta 中 FAN 算法, 包含蕴含、线确认、回溯等重要过程) 得不到现有图计算系统的支持. 现有图计算系统主要研究如何高效支持单个图分析任务的有效执行. 然而, ATPG 算法处理每个故障都是一个图分析任务. 这意味着, 将 ATPG 算法转化为图算法后, 其需要执行大量的并发图分析任务 (即, 它需要故障列表中的故障全部处理完成才算结束). 因此, 将 ATPG 算法转化为图算法后, 其高效执行面临下述挑战: (1) 在现在图计算系统上, 大量并发图分析任务会独立地存储和访问它们各自的电路图数据, 从而存在大量冗余数据存储和访问, 导致内存消耗大和数据访问开销高的问题. (2) 大量并发图分析任务的非规则数据访问, 导致与数据访问相关的资源 (例如 cache 和内存带宽) 竞争激烈, 进一步致使数据访问开销高, 大量核处于空闲状态. (3) 大量并发图分析任务 (例如对于实验中的电路图 w1∼w8, 故障列表的大小是 258200∼424800, ATPG 算法对应的并发图分析任务的数量会更多) 调度开销大并且难以有效调度.

在现有图计算系统上执行电路测试算法面临以下几个性能瓶颈.

(1) 不合理的分区策略导致冗余磁盘 I/O. 一般来说, 图算法通过沿路径传播活跃顶点的状态值来更新邻接顶点状态, 最终达成收敛 [44]. 在现实图中, 因为环的存在, 顶点的状态可能在环上反复传播多次才能达成收敛条件, 这意味着某些顶点会被反复处理. 电路图不同于一般现实图, 它是单向图. 图算法运行在电路图上的时候, 同样需要对顶点状态进行传播. 因为不存在环, 每个顶点只需要对状态传播一次. 这使得图算法对顶点的处理有绝对的顺序性, 这种顺序性同状态传播的单向路径一致. 在单次图算法运行过程中, 完成状态传播的顶点不再需要进行二次处理, 是无用顶点. 目前的图计算领域还没有针对单向图顶点处理的顺序性进行分区的图划分策略. 现有的划分方式很可能导致同一条路径上不相邻的顶点被划分到一起, 在图处理时这种分区会从磁盘中反复被加载, 造成冗余磁盘 I/O 开销.

(2) 单个图分析任务有效并行度低. 在图计算中, 同一条路径上的顶点无法进行有效的并行处理. 这是因为有关路径后端顶点的处理, 图计算引擎必须等接收到前端顶点传播的状态后才能开始, 否则只能对后端顶点做无效处理. 因此, 图计算引擎只能并行处理不同路径上的顶点. 对于现实图, 它顶点的度往往满足幂律分布, 而顶点每一条边都是不同路径的一部分, 所以一般的图算法往往能以很高的并行度运行. 而电路图的边的分布在各顶点间相对平均, 不具备幂律性, 这使得电路图的路径相对于现实图要少很多. 除此之外, 有一些电路测试算法只需要对电路的很小部分进行处理. 如果将这些测试算法以图算法的形式描述, 那么整个算法执行期间可能就只处理少数几条路径, 根本无法充分利用上并行计算资源. 为了达到较高并行度, 必须考虑多个图分析任务同时并发处理的方案, 但这样会面临新的问题.

(3) 并发图分析任务执行时产生大量冗余存储开销和数据访问开销. 有很多电路测试算法需要大量反复地执行, 它们在转化为图算法后相当于大量的图分析任务. 而且上面提到单个图分析任务存在有效并行度低的问题, 这两点使得我们不得不考虑进行图分析任务的并发处理. 现有的图计算系统大多专注于如何高效执行单个图分析任务的问题. 这些系统在支持多个图分析任务并发执行时, 没有任

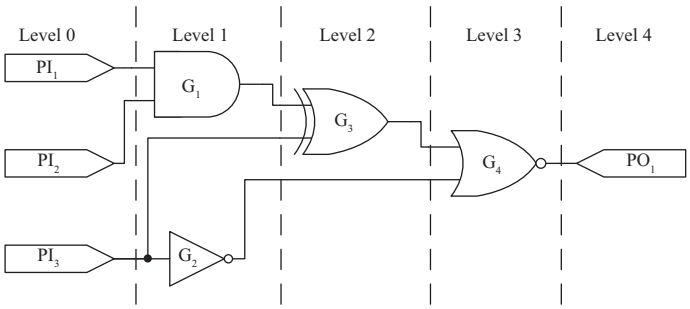

图 4　电路中逻辑门的逻辑级
Figure 4　Logic level of the logic gate in the circuit

何优化策略. 在并发处理时, 这些系统为每个单独的图分析任务维护一个图结构数据的副本, 而不考虑这些图任务是否在相同的图结构上执行. 这种粗糙的并发处理面临低效的缓存 (cache)、低效的内存和低效的数据访问通道共享使用, 这导致数据访问瓶颈和性能干扰问题, 还会造成大量的额外存储开销 [44, 46∼52].

6　我们的工作进展

为了提高电路测试算法的并行性, 我们设计出专门面向 ATPG 的单机图计算系统. 此系统能够将大量重复的测试图算法以并发图分析任务的形式进行处理, 并且这些并发作业在运行时共享同一份图结构数据. 一方面, 多任务并发处理解决了单个测试算法在转化为图算法后有效并行度不足的问题. 另一方面, 此系统能够通过高效的同步控制机制实现并发任务对单一图结构数据副本的共享, 这样便于大大减少存储开销和数据访问成本. 除此之外, 在预处理阶段, 此系统能够处理基于逻辑级的图划分策略, 规则化处理过程对磁盘上分区的加载, 避免了额外的磁盘 I/O.

6.1　相关概念定义

本小节给出图计算和电路测试相关的基本概念和符号解释.

定义1 (图分区)　给定一个有向图 $G=(V,E)$, 其中 V 表示有向图 G 中图顶点的集合, E 表示有向图 G 中边的集合 [17]. 图分区将有向图 G 的顶点集合 V 划分为 n 个互不相交的顶点子集 [17, 80, 81], 记作 $V_1, V_2, \ldots, V_i, \ldots, V_n$, $(1 \leqslant i \leqslant n)$. 分区 P_i 的顶点集合 V_{P_i} 为有向图 G 的第 i 个顶点子集 V_i, P_i 的边的集合$E_{P_i}=\{e(u,v)\in E|u\in V_{P_i}\}$.

定义2 (逻辑级)　在组合逻辑电路 C 中, 每个逻辑门的逻辑级被定义为逻辑值从输入端传输到该逻辑门所需要经过的逻辑门数 (包含它本身) 的最大值. 广义的逻辑级定义同样适用于输入端和输出端, 显然, 所有输入端的逻辑值均为 0. 图 4 中给出了示例电路所有逻辑门和输入输出的逻辑级.

定义3 (并发 ATPG 任务)　ATPG 中包含测试生成和故障模拟两个主要过程, 这两个过程都是由独立的子任务组成. 对于测试生成, 子任务是指不同故障的测试向量生成. 对于故障模拟, 子任务是指同一测试向量对不同故障的故障模拟. 上述两类子任务可以各自地并发执行, 在本文中统一称为并发 ATPG 任务.

6.2　系统框架和功能模块

通常, 图计算任务所需的数据类型可以分为两类. 一类是图结构数据, 即由 $G=(V,E)$ 表示的图

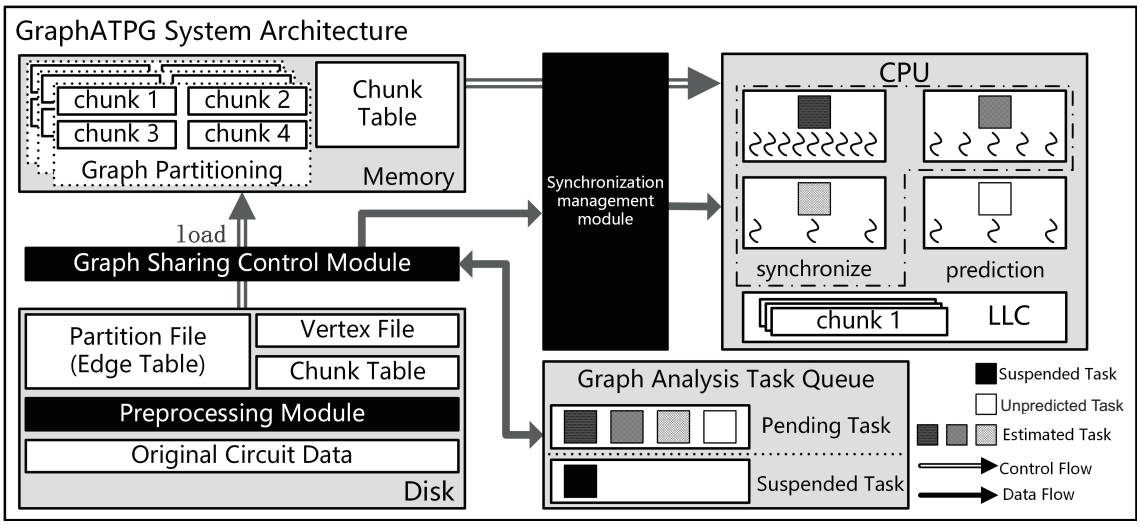

图 5 GraphATPG 系统架构
Figure 5 Architecture of the GraphATPG system

拓扑. 另一类是状态数据, 状态数据指示着图中顶点或边的状态. 对于不同的图分析任务, 状态数据不同. 比如对于 PageRank 算法 [44, 85], 要使用 PR 值 (网页的排名分) 作为顶点的状态数据. 而对于连通分量算法 [44], 每个顶点都需要一个分量 ID 作为状态数据. 在图处理过程中, 图计算任务需要通过遍历图结构数据对顶点和边执行特定操作以更新状态数据, 直到计算结果达到收敛条件. 一般来说, 现有的图计算系统都是将图结构数据和状态数据分开存储的 [12, 44]. 对于转化为图算法后的电路测试过程, 往往有大量的图分析任务需要执行, 而这些图分析任务在相同的图数据结构上执行. 本文提出的系统可以充分利用这些图分析任务访问图结构数据时的时空关联性, 使图结构数据被并发执行的图分析任务共享. 图 5 给出了本文 GraphATPG 的系统架构. 它由 3 个主要模块组成: 预处理模块、图共享控制模块和同步管理模块, 下面将介绍这些模块的作用.

预处理模块. 在电路设计领域, 通常使用网表文件来描述电路的拓扑结构. 网表文件中包含电路中器件的标示、封装、连接关系, 但无法直接用于图计算系统. 因此, 在图处理之前, 必须将原来的网表文件转换为 GraphATPG 所需要的顶点文件和边列表文件. 其中, 顶点文件为电路中每个 PI、PO、逻辑门和扇出节点分配不同的顶点 ID, 并指示出对应顶点 ID 的类型. 如果顶点类型为逻辑门, 会给出具体的逻辑门类型. 边列表文件则以二元组的形式存储电路图中的每条有向边. 在此之后, GraphATPG 读取顶点文件和边列表文件, 以流的形式处理边列表以计算各顶点的逻辑级. GraphATPG 通过计算得出的逻辑级对电路图进行分区, 每个分区在磁盘上都有单独的分区文件. 具体来说, GraphATPG 将逻辑级划分成 n 个连续的区间, 所有边根据目的顶点的逻辑级划分到对应区间组成分区, 所有分区中的边的数量大体相同. 除此之外, GraphATPG 还会生成一个元文件, 其中包含电路图的全局信息. 这种基于逻辑级的分区能够规则化图处理过程中对分区的 I/O 行为, 减少数据访问开销. 同时, 为了达到细粒度同步的目的, GraphATPG 会将分区进一步逻辑划分为块. 划分后的单个块可以整个加载到末级缓存 (last level cache, LLC) 上, 因此有很好的缓存局部性.

图共享控制模块. 在预处理模块中, 电路图根据逻辑级被划分为多个分区, 每个分区对应磁盘上的一个分区文件. 图共享控制模块通过控制加载分区文件的顺序和选择特定图分析任务进行并发处理来确保分区中的图结构数据能够被大部分图分析任务所共享. 电路图与传统图计算中所处理的一般图

有很多不同之处. 作为有向图, 电路图是不存在环的, 也就是说电路图是单向图, 而且电路图中各顶点的度数分布也不符合幂律性. 一般的图算法在执行过程中, 通常需要对某几个度数很高的顶点进行反复处理. 一是因为度数高的顶点受其他顶点的影响的概率很高. 二是因为图中存在环, 顶点的状态传播可能回到自身. 而对于测试算法, 顶点状态的传播通常是按照逻辑级的升序或降序来单向地进行: 它从特定逻辑级的顶点开始, 往相邻的逻辑级传播, 只有处理完当前逻辑级的所有顶点才能开始对下一逻辑级的顶点进行有效处理. 因此, 基于逻辑级的分区需要使得单个图分析任务对分区的加载顺序在一定程度上可以预测. 图共享控制模块会优先对起始逻辑级相同, 并且顶点状态传播方向相同的图分析任务进行并发处理. 这些图分析任务不仅有加载大量相同分区的需求, 而且对于分区的加载在时序性上也是强一致的, 这使得加载的分区能够被大部分并发任务处理. 在所有并发任务处理完当前分区相关状态数据后, 图共享控制模块统计所有并发任务需要处理的下一逻辑级并据此决定所需要加载的下一个分区. 如果某一并发任务无需对加载的分区进行处理, 那么它将会暂时被挂起, 等待之后有效分区的加载. 图共享控制模块使得 GraphATPG 只需要在内存中维护图结构数据的单一副本就能进行并发图任务分析, 这大大减少了存储开销和磁盘 I/O 传输开销.

同步管理模块. 图共享控制模块使得并发图分析任务以一致的顺序访问图分区, 这些分区在预处理过程被进一步细分为块. 尽管各个图分析任务的活跃顶点不完全相同, 但这些顶点可能在同一个分块中, 分块在载入 LLC 时能够被各任务共享. 由于电路图拓扑的不规则性, 各个图分析任务在处理共享分块时, 分块内的活动边数量不同, 因此计算的复杂度也并不相同. 这使得各图分析任务共享的块会不规则且反复地流到 LLC 中, 导致冗余的数据访问成本. 为了解决这一问题, 同步管理模块采用了一种高效的细粒度同步方法, 充分利用图分析任务之间的时空关联性. 此模块通过细粒度同步图分析任务之间的访问路径, 使得共享的图结构数据块能够规则化地流入 LLC 中. 具体来说, 同步管理模块会在每次迭代之前确定不同任务在每个块上的工作负载. 然后, 同步管理模块根据这些任务的工作量分配相应的计算资源以供它们并发执行, 从而以较低的代价实现它们对分区遍历的同步. 通过这种方式, 每个共享块只需要加载到 LLC 中一次, 显著降低了数据访问成本.

GraphATPG 的基本工作流程如下所述. 预处理模块分 4 步将原始电路数据处理为电路分析平台所需要的数据形式. 第 1 步, 将原始电路数据中的电路拓扑信息转化为邻接矩阵的形式. 第 2 步, 考虑到电路图的稀疏性, 预处理模块会将邻接矩阵采用压缩格式存储, 如 CSR 格式. 具体过程是按顶点升序对邻接矩阵的行进行遍历, 把非 0 项的列索引按遍历顺序存入一个边列表, 并用一个偏移数组指示每个顶点在边列表中对应的项. 第 3 步, 根据式 (4) 计算出分区数 P:

$$\frac{|V|}{P} \times (D_{\mathrm{vp}} + |J| \times D_{\mathrm{vs}} + 2 \times D_{\mathrm{e}}) \leqslant |\mathrm{Mem}|. \tag{4}$$

其中, $|V|$ 为图顶点数, $|V|/P$ 则代表 1 个分区中的顶点数, D_{vp} 为顶点属性数据大小 (包括逻辑门类型等), $|J|$ 为用户设定的最高并发任务数, D_{vs} 为顶点状态数据大小, D_{e} 为边数据大小 (分区中边的数量接近顶点数的二倍). $|\mathrm{Mem}|$ 为机器物理内存大小. 分区数 P 应该取满足式 (4) 的最小整数值. 在得出分区数 P 后, 将电路图的所有边根据其源顶点索引的逻辑级数分成大小相等的 P 份, 每份对应一个分区. 可以通过一个分区表记录每个分区的首顶点索引. 第 4 步, 计算出逻辑块大小阈值 $|C|$, 将所有分区在逻辑上划分为块, 块信息存储在分块表 (chunk table) 中, 包括块索引、块所在分区索引、起始顶点索引和结束顶点索引. 在原始电路数据的预处理完成后, 用户可通过数据分析平台提供的一般图计算框架 API, 将各种基于拓扑的电路分析算法实现为迭代图算法. 这些算法需要在不同的初始化条件下大量重复执行, 每一次执行都相当于一个单独的图分析任务. 因为是对同一个电路进行处理, 这

些图分析任务依赖于完全相同的图结构数据. 它们在电路分析平台并发地执行, 并且通过共享内存的方式实现对同一份图结构数据副本的访问.

图共享控制模块通过实时分析各分区加载的优先级, 将优先级最高的分区加载到内存中供并发任务处理. 具体实施时, 图共享控制模块会维护一个并发任务列表 parallel_job_list 和一个分区状态表 partition_state_table. parallel_job_list 中存储了各个任务的进程 id、进程状态和待处理分区索引等信息, 用户通过定义 MAX_JOB_NUM 的值来限制 parallel_job_list 的大小, 即最大并发任务数. 当 parallel_job_list 中任务数没有达到最大值时, 由用户提供新的并发任务. partition_state_table 中存储分区索引、分区相关任务数、分区相关任务链表的头结点指针. 注意, 一个任务 j 为分区 p 的相关任务, 说明任务 j 待处理的分区为 p. 在初始化过程, 所有并发任务被挂起, 图共享控制模块遍历 parallel_job_list, 获取每个任务的待处理分区索引, 然后更新 partition_state_table 中分区索引相同的表项. 即每有一个相关任务, 将表项中分区相关任务数增加 1, 并将任务进程 id 作为链表新结点插入相关任务链表尾部. 接着, 图共享控制模块获取相关任务数最大的分区索引, 将该分区图结构数据从磁盘加载到内存中, 把该分区相关任务的进程恢复, 并且在 partition_state_table 中将加载分区索引对应的相关任务数清零, 链表结点清空. 当相关任务处理完当前分区内所有活跃顶点后, 可以获取下一轮迭代所需要处理的活跃顶点, 并据此得到待处理的分区. 具体来说, 通过位图来指示任务活跃顶点, 然后通过活跃顶点的索引所在的分区索引区间判断该任务下一轮的待处理分区. 每有一个并发任务对当前加载分区的处理结束, 就更新 parallel_job_list 中该并发任务的待处理分区, 再更新 partition_state_table 中与待处理分区索引相同的表项, 然后挂起该任务. 如果该任务达成最终收敛, 则从 parallel_job_list 中删除该任务对应条目, 等待用户提供新的并发任务. 新的并发任务在初始化时必须由用户提供第 1 轮迭代的活跃顶点, 可以据此来获取新任务的第 1 个待处理分区, 并更新表 partition_state_table. 当所有并发任务都被挂起或达成收敛后, 图共享处理模块结束对当前分区的处理, 从磁盘中加载 partition_state_table 中相关任务数最大的分区到内存中, 重复上述过程, 直到列表 parallel_job_list 为空.

同步管理模块负责将各活动并发任务对分区数据的加载进行同步. 具体实施时, 各并发任务以逻辑块为单位, 将数据加载到 LLC 中进行处理. 每当加载一个逻辑块到 LLC 中, 各个并发任务通过位图获取它们在该逻辑块中要处理的活跃顶点. 同步管理模块根据各个并发任务在当前逻辑块上工作负载, 按比例地分配计算资源, 使得各并发任务能够在同一时间完成对当前逻辑块的处理. 之后, 同步管理模块开始加载下一个活跃块到 LLC 中, 直到所有活跃块处理完毕.

故障十分稀疏也不会限制我们系统并行度. 因为当任务数少于核的数量时, 本文系统的负载均衡策略会根据空闲核的数量将最慢的任务的待处理数据 (其待处理的电路图数据) 均分成多份, 交由这些空闲核并行处理. 例如, 就算只有一个任务, 本文系统会将该任务待处理的数据均分成多份交由空闲核并行处理, 从而保证系统的并行度. 通常 ATPG 算法的任务数量 (等于被测试的故障列表中的故障数量, 例如对于实验中的电路图 w1∼w8, 故障列表的大小是 258200∼424800, 任务数量会更多) 远大于核的数量 (例如我们实验平台的核数是 28 核). 因此, 在我们系统中核大多数情况都处于工作状态 (可能只在最后时刻有少量核处于空闲状态). 在本文系统中, 每个核负责处理一个任务的部分或全部电路图数据的处理 (需要注意的是, 当任务数少于核的数量时, 系统的负载均衡策略会根据空闲核的数量将最慢的任务的待处理电路图数据均分成多份, 交由这些空闲核并行处理). 当某个核处理完某个任务的数据时, 该任务相关的数据会被新分配的任务的数据替换. 这也意味着, 在本文系统中, 任务的数量最多也只会和核的个数相同数量. 也就是说, 最多只有与核个数相同数量的任务的数据在内存中 (例如我们平台有 28 核, 因此只有 28 个任务的数据被放在内存中), 不会占用大量的内存空间.

我们在一台 Linux 服务器 (28-core Intel Xeon(R) Gold 5117 CPU 和 256 GB 内存, CPU 的频率

表 2 ATPG 的实验结果
Table 2 Experimental results of ATPG

Circuit name	Vertices	Edges	Faults	ATPG runtime			Memory overhead		
				Atalanta (s)	GP (s)	Reduction (%)	Atalanta (MB)	GP (MB)	Reduction (%)
w1	68600	132200	258200	969	381	60.68	76.864	24.132	68.60
w2	74800	144200	282000	1018	408	59.92	78.189	24.588	68.55
w3	81000	156200	305800	1054	426	59.58	79.686	25.545	67.94
w4	87200	168200	329600	1171	446	61.91	81.463	26.184	67.86
w5	93400	180200	353400	1091	519	52.43	82.952	27.449	66.91
w6	99600	192200	377200	1122	503	55.17	84.337	27.792	67.05
w7	105800	204200	401000	1196	532	55.52	85.197	28.509	66.54
w8	112000	216200	424800	1221	544	55.45	85.832	29.146	66.04

是 2 GHz) 上, 对 8 个电路 w1∼w8 进行了实验, 电路是组合逻辑网络, 来源于华为海思半导体公司, 电路的特征如表 2 所示. 用我们当前系统 (graph processing based approach, GP) 得到的实验结果和 Atalanta 得到的结果如表 2 所示, 结果表明本文方法缩短了运行时间和减少了内存开销.

7 未来的挑战和相关研究方向

为了进一步提高测试的速度、减少内存开销, 以及提高可扩展性, 可以通过优化图计算系统解决上述问题. 我们可考虑以下几个方面.

7.1 传统架构上的软件系统优化

在现有图计算软件系统中, 并发 ATPG 任务互不感知数据访问的空间/时间关联性, 并且不同图计算任务的图顶点状态沿着复杂多变的依赖进行传递. 因此, 并发 ATPG 任务在执行中会存在大量冗余且不规则的数据访问, 这使得他们面临着严重的数据访问瓶颈和底层硬件资源难以有效利用等挑战, 最终导致现有软件系统在支持并发 ATPG 任务执行时吞吐率低. 为此, 如何设计面向 ATPG 的图计算软件系统来充分挖掘和利用并发 ATPG 任务之间在运行时的数据访问关联性和图顶点状态之间的依赖关联性, 并透明有效地支持并发 ATPG 任务在底层硬件平台上的执行是亟需解决的问题. 由于电路图的单向性和电路算法特性, 图计算过程对图数据的访问有严格的顺序性和可预测性, 但现有图划分策略没有充分利用这一特性, 导致冗余的数据访问开销. 可挖掘 ATPG 算法图顶点更新顺序和电路图拓扑之间的相关性, 根据相关性对图数据进行划分, 保证图分区中数据访问的局部性. 在图划分时将处理顺序相近的顶点划分到同一分区中, 避免图分区中冗余数据的反复加载. 另外, 可对各任务活跃图数据进行高效识别, 对并发任务数据访问相似性挖掘, 自适应地对图数据进行动态重划分, 保证较小的随机磁盘访问的同时减少冗余图数据加载, 还可以研究图结构感知的图数据缓存管理.

7.2 基于硬件加速器的技术

随着新兴硬件 (例如, GPU, FPGA, 以及 ASIC) 的不断发展, 这给如何高效支持并发 ATPG 任务执行带来了机遇和挑战. 虽然 GPU 通过提供大规模并行计算资源来获得高性能, 但是并发 ATPG 任务之间不规则的数据访问导致大量冗余访问开销, 从而限制了并发 ATPG 任务计算的并行度. 为此, 如何设计有效的执行机制来高效利用 GPU 的高并行性是提高并发 ATPG 任务计算吞吐率的重要途径. 例如, 我们可以有效地规则化和加快图顶点状态传递、解耦合数据依赖、保证高效的联合数据访问和负载均衡, 从而使得并发 ATPG 任务能够充分利用底层并行计算资源, 获得高吞吐率. 此外, 虽然 FPGA 和 ASIC 能够为并发 ATPG 任务执行提供专用的存储子系统和处理单元来有效提高计算的效率, 但是它们面临着片上存储资源有限等挑战. 因此, 我们可以考虑如何利用各并发 ATPG 任务之

间数据访问的空间局部性和时间局部性, 实现图结构数据和图顶点状态数据的高效共享和合并访问, 充分提高片上存储资源和内存带宽的利用率.

7.3 基于新兴存储硬件的优化

大量新兴存储硬件的涌现也为如何有效降低并发 ATPG 任务的数据访问开销提供了机遇. NVM, HMC 和 ReRAM 等新兴存储硬件能够有效减少数据移动与通信消耗. 由于图计算是一种数据访问密集型的应用, 因此可以通过有效利用并发 ATPG 任务的数据访问关联性等特征和充分挖掘 NVM, HMC 与 ReRAM 的硬件特性, 从而极大地降低并发 ATPG 任务的数据访问开销. 并且, 现实世界的图往往规模巨大而采用分布式集群的方式来处理, 为此, 可以通过有效地利用 RDMA 等新兴硬件来减少并发 ATPG 任务在机器间的通信开销和降低其通信延迟. 以上新兴硬件都为高效执行并发 ATPG 任务提供了新的可能. 并发图处理系统无法感知底层的内存异构性, 其读写性能不对称性仍会导致数据访问开销高等问题. 基于 RDMA 的远程 NVM 数据读写仍是数据访问瓶颈. 可对面向异构内存的图数据存储机制进行研究: 将不同的数据根据其访存特征存储在合适的内存器件中; 在运行时监测访存变化、动态迁移图数据、保证负载均衡等. 可监测数据冷热, 将具有高访问频率的数据迁移至远程 DRAM 中, 加速基于 RDMA 的远程数据的访问. 也可探索异步 RDMA 写操作来进一步减少写延迟, 从而降低数据访问开销.

8 总结

ATPG 算法的高效执行可以节省 VLSI 测试和验证时间以及测试成本. 现有的 ATPG 方法普遍存在负载不均衡、并行策略单一、存储开销大和数据局部性差等问题. 由于图计算具有高效并行度和高扩展性等优点, 因此本文对图计算在 ATPG 中的应用进行了探究. 本文提出了 ATPG 算法转化为图算法的方法, 指出了图计算应用在组合电路 ATPG 中面临的挑战, 并提出了我们设计的面向 ATPG 的单机图系统. 同时本文也对图计算系统支持 ATPG 未来所面临的挑战和研究方向进行了探讨. 本文为电路测试提供了新方法并进行了探索, 有助于相关研究人员设计高效的并行 ATPG 方案, 从而加速芯片的设计、验证过程并且节省测试成本. 未来我们也将研发 EDA 一体机, 高效支持 EDA 算法的执行, 希望相对于现有商业软件获得一个数量级以上的性能提升.

参考文献

1 Liu P X. Research of parallel ATPG algorithm and prototype system design. Dissertation for Ph.D. Degree. Changsha: National University of Defense Technology, 2002 [刘蓬侠. 并行 ATPG 算法理论与原型系统设计技术研究. 博士学位论文. 长沙: 国防科学技术大学, 2002]

2 Chen X Y. Research on optimization of diagnostic test pattern set and fault test pattern set. Dissertation for Master's Degree. Changchun: Jilin University, 2020 [陈晓艳. 诊断测试集和故障测试集优化问题研究. 硕士学位论文. 长春: 吉林大学, 2020]

3 Drechsler R, Fey G. Automatic test pattern generation. In: Lecture Notes in Computer Science. Berlin: Springer-Heidelberg, 2006

4 Liu X. Research on faulty test pattern generation methods for digital circuits. Dissertation for Ph.D. Degree. Wuhan: Huazhong University of Science and Technology, 2004 [刘歆. 数字电路的故障测试模式生成方法研究. 博士学位论文. 武汉: 华中科技大学, 2004]

5 Wolf J M, Kaufman L M, Klenke R H, et al. An analysis of fault partitioned parallel test generation. IEEE Trans Comput-Aided Des Integr Circuits Syst, 1996, 15: 517–534

6 Ramkumar B, Banerjee P. ProperTEST: a portable parallel test generator for sequential circuits. IEEE Trans Comput-Aided Des Integr Circuits Syst, 1997, 16: 555–569

7 Patil S, Banerjee P. Performance trade-offs in a parallel test generation/fault simulation environment. IEEE Trans Comput-Aided Des Integr Circuits Syst, 1991, 10: 1542–1558

8 Patil S. Parallel algorithms for test generation and fault simulation. Dissertation for Ph.D. Degree. Urbana-Champaign: University of Illinois, 1991

9 Moue T, Fujii T, Fujiwara H. Performance analysis of parallel test generation for combinational circuits. IEICE Trans Info Sys, 1996, 79: 1257–1265

10 Fujiwara H, Inoue T. Optimal granularity and scheme of parallel test generation in a distributed system. IEEE Trans Parallel Distrib Syst, 1995, 6: 677–686

11 Corno F, Prinetto P, Rebaudengo M, et al. GATTO: a genetic algorithm for automatic test pattern generation for large synchronous sequential circuits. IEEE Trans Comput-Aided Des Integr Circ Syst, 1996, 15: 991–1000

12 Wang J, Zhang L, Wang P Y, et al. Memory system optimization for graph processing: a survey. Sci Sin Inform, 2019, 49: 295–313 [王靖, 张路, 王鹏宇, 等. 面向图计算的内存系统优化技术综述. 中国科学: 信息科学, 2019, 49: 295–313]

13 Fujiwara H, Toida S. The complexity of fault detection problems for combinational logic circuits. IEEE Trans Comput, 1982, 31: 555–560

14 Liu G X, Ge H T, Chen K X. Principle of automatic test vector generation technique for gate-level circuits. J Zhejiang Univ, 2006, 1: 52–57 [刘观生, 葛海通, 陈偕雄. 门级电路自动测试向量生成技术原理. 浙江大学学报, 2006, 1: 52-57]

15 Nagle H T, Roy S C, Hawkins C F, et al. Design for testability and built-in self test: a review. IEEE Trans Ind Electron, 1989, 36: 129–140

16 Williams T W, Parker K P. Design for testability — a survey. Proc IEEE, 1983, 71: 98–112

17 Eichelberger E B, Williams T W. A logic design structure for LSI testing. In: Proceedings of the 14th Design Automation Conference, 1977. 462–468

18 Funatsu S, Wakatsuki N, Arima T. Test generation systems in Japan. In: Proceedings of the 12th Design Automation Conference, 1975. 114–122

19 Koenemann B, Mucha J, Zwiehof G. Built-in logic block techniques. In: Proceedings of International Test Conference, 1979. 37–41

20 Saluja N, Gulati K, Khatri S P. SAT-based ATPG using multilevel compatible don't-cares. ACM Trans Des Autom Electron Syst, 2008, 13: 1–18

21 Bell R H, Klenke R H, Aylor J H, et al. Results of a topologically partitioned parallel automatic test pattern generation system on a distributed memory multiprocessor. In: Proceedings of Application Specific Integrated Circuits Conference, 1992. 380–384

22 Roth J P. Diagnosis of automata failures: a calculus and a method. IBM J Res Dev, 1966, 10: 278–291

23 Goel P. An implicit enumeration algorithm to generate tests for combinational logic circuits. IEEE Trans Comput, 1981, 30: 215–222

24 Fujiwara H, Shimono T. On the acceleration of test generation algorithms. IEEE Trans Comput, 1983, 32: 1137–1144

25 Sellers F F, Hsiao M Y, Bearnson L W. Analyzingerrors with the Boolean difference. IEEE Trans Comput, 1968, 17: 676–683

26 Shi J, Fey G, Drechsler R, et al. PASSAT: efficient SAT-based test pattern generation for industrial circuits. In: Proceedings of IEEE Computer Society Annual Symposium on VLSI, 2005. 212–217

27 Tafertshofer P, Ganz A, Henftling M. A sat-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists. In: Proceedings of IEEE/ACM International Conference on Computer-Aided Design, 1997. 648–655

28 Safarpour S, Veneris A, Drechsler R, et al. Managing don't cares in boolean satisfiability. In: Proceedings of Design, Automation and Test in Europe Conference (DATE), 2004. 260–265

29 Ulrich E G, Baker T. Concurrent simulation of nearly identical digital networks. Computer, 1974, 7: 39–44

30 Kim K, Saluja K K. On fault detection problem in concurrent fault simulation for synchronous sequential circuits. In: Proceedings of IEEE VLSI Test Symposium, 1982. 125–130

31 Lee D H, Reddy S M. On efficient concurrent fault simulation for synchronous sequential circuits. In: Proceedings the

29th ACM/IEEE Design Automation Conference, 1992. 327–331
32 Saab D G, Hajj I N, Rahmeh J T. Parallel-concurrent fault simulation. In: Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1989. 298–301
33 Abramovici M, Menon P R, Miller D T. Critical path tracing: an alternative to fault simulation. IEEE Des Test Comput, 1984, 1: 83–93
34 Antreich K J, Schulz M H. Accelerated fault simulation and fault grading in combinational circuits. IEEE Trans Comput-Aided Des Integr Circ Syst, 1987, 6: 704–712
35 Maamari F, Rajski J. A fault simulation method based on stem regions. In: Proceedings of IEEE International Conference on Computer-Aided Design (ICCAD-89) Digest of Technical Papers, 1988. 170–173
36 Lee H K, Ha D. An efficient, forward fault simulation algorithm based on the parallel pattern single fault propagation. In: Proceedings of International Test Conference, 1991. 946–955
37 Gouders N, Kaibel R. Paris: a parallel pattern fault simulator for synchronous sequential circuits. In: Proceedings of IEEE International Conference on Computer-Aided Design Digest of Technical Papers, 1991. 542–545
38 Nair R, Dong S H. Vision: an efficient parallel pattern fault simulator for synchronous sequential circuits. In: Proceedings of the 13th IEEE VLSI Test Symposium, 1995. 221–226
39 Harel D, Sheng R, Udell J. Efficient single fault propagation in combinational circuits. In: Proceedings of International Conference on Computer Aided Design, 1987. 2–5
40 Lai L, Tsai K H, Li H. GPGPU-based ATPG system: myth or reality? IEEE Trans Comput-Aided Des Integr Circ Syst, 2020, 39: 239–247
41 Gulati K, Khatri S P. Towards acceleration of fault simulation using Graphics. In: Processing of the 45th ACM/IEEE Design Automation Conference, 2008. 822–827
42 Li M, Hsiao M S. 3-D parallel fault simulation with GPGPU. IEEE Trans Comput-Aided Des Integr Circ Syst, 2011, 30: 1545–1555
43 Liao K Y, Hsu S C, Li J C M. GPU-based N-detect transition fault ATPG. In: Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference (DAC), 2013. 1–8
44 Zhang Y, Liao X F, Jin H, et al. DepGraph: a dependency-driven accelerator for efficient iterative graph processing. In: Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2021. 371–384
45 Lee H K, Ha D S. On the generation of test patterns for combinational circuit. 2013. https://github.com/hsluoyz/Atalanta
46 Zhang Y, Liao X F, Jin H, et al. DepGraph: an efficient path-based iterative directed graph processing system on multiple gpus. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19), 2019. 601–614
47 Zhang Y, Liao X, Gu L, et al. AsynGraph: maximizing data parallelism for efficient iterative graph processing on GPUs. ACM Trans Archit Code Optim, 2020, 17: 1–21
48 Zhang Y, Zhao J, Liao X, et al. CGraph: a distributed storage and processing system for concurrent iterative graph analysis jobs. ACM Trans Storage, 2019, 15: 1–26
49 Zhao J, Zhang Y, Liao X F, et al. GraphM: an efficient storage system for high throughput of concurrent graph processing. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19), 2019. 1–14
50 Zhang Y, Liao X, Jin H, et al. FBSGraph: accelerating asynchronous graph processing via forward and backward sweeping. IEEE Trans Knowl Data Eng, 2018, 30: 895–907
51 Zhang Y, Liao X, Shi X, et al. Efficient disk-based directed graph processing: a strongly connected component approach. IEEE Trans Parallel Distrib Syst, 2018, 29: 830–842
52 Zhang Y, Liao X F, Jin H, et al. CGraph: a correlations-aware approach for efficient concurrent iterative graph processing. In: Proceedings of USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18), 2018. 441–452
53 Gui C Y, Zheng L, He B, et al. A survey on graph processing accelerators: challenges and opportunities. J Comput Sci Technol, 2019, 34: 339–371
54 Sabet A H N , Zhao Z J, Gupta R. Subway: minimizing data transfer during out-of-gpu-memory graph processing.

In: Proceedings of the 15th EuroSys Conference (EUROSYS'20), 2020
55 Jin H, Shi X H. Big Data Processing. Beijing: Higher Education Press, 2018 [金海, 石宣化. 大数据处理. 北京: 高等教育出版社, 2018]
56 Shun J L, Blelloch G E. Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2013. 135–146
57 Nguyen D, Lenharth A, Pingali K. A lightweight infrastructure for graph analytics. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, 2013. 456–471
58 Sundaram N, Satish N, Patwary M M A, et al. GraphMat. Proc VLDB Endow, 2015, 8: 1214–1225
59 Zhang K Y, Chen R, Chen H B, et al. Numa-aware graph-structured analytics. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2015. 183–193
60 Zhang Y, Liao X, Jin H, et al. HotGraph: efficient asynchronous processing for real-world graphs. IEEE Trans Comput, 2017, 66: 799–809
61 Kyrola A, Blelloch G, Guestrin C. GraphChi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, 2012. 31–46
62 Ko S, Han W S. Turbograph++: a scalable and fast graph analytics system. In: Proceedings of ACM SIGMOD International Conference on Management of Data, 2018. 395–410
63 Cheng J F, Liu Q, Li Z G. Venus: vertex-centric streamlined graph computation on a single PC. In: Proceedings of IEEE International Conference on Data Engineering, 2015. 124–134
64 Roy A, Mihailovic I, Zwaenepoel W. X-stream: edge-centric graph processing using streaming partitions. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, 2013. 472–488
65 Yuan P P, Zhang W Y, Xie C F, et al. Fast iterative graph computation: a path centric approach. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, 2014. 401–412
66 Zhu X, Han W, Chen W. GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: Proceedings of Usenix Conference on Annual Technical Conference, 2015. 375–386
67 Chi Y Z, Dai G H, Wang Y, et al. NXgraph: an efficient graph processing system on a single machine. In: Proceedings of IEEE International Conference on Data Engineering, 2016. 409–420
68 Cheng R, Hong J, Kyrola A, et al. Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems, 2012. 85–98
69 Han W T, Miao Y S, Li K W, et al. Chronos: a graph engine for temporal graph analysis. In: Proceedings of the 9th European Conference on Computer Systems, 2014. 1–14
70 Malewicz G, Austern M H, Bik A J, et al. Pregel: a system for large-scale graph processing. In: Proceedings of ACM SIGMOD International Conference on Management of Data, 2010. 135–146
71 Yan D, Cheng J, Lu Y, et al. Blogel. Proc VLDB Endow, 2014, 7: 1981–1992
72 Low Y, Bickson D, Gonzalez J, et al. Distributed GraphLab. Proc VLDB Endow, 2012, 5: 716–727
73 Gonzalez J E, Low Y C, Gu H J, et al. PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), 2012. 17–30
74 Xie C N, Chen R, Guan H B, et al. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In: Proceedings of ACM Sigplan Symposium on Principles and Practice of Parallel Programming, 2015. 194–204
75 Merrill D, Garland M, Grimshaw A. Scalable GPU graph traversal. SIGPLAN Not, 2012, 47: 117–128
76 Wang Y Z H, Davidson A A, Pan Y C, et al. Gunrock: a high-performance graph processing library on the GPU. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15), 2015. 265–266
77 Han W, Mawhirter D, Wu B, et al. Graphie: large-scale asynchronous graph traversals on just a GPU. In: Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT'17), 2017. 233–245
78 Zhou S, Kannan R, Prasanna V K, et al. HitGraph: high-throughput graph processing framework on FPGA. IEEE Trans Parallel Distrib Syst, 2019, 30: 2249–2264
79 Dai G, Huang T, Chi Y, et al. ForeGraph: exploring large-scale graph processing on multi-FPGA architecture. In: Proceedings of ACM International Symposium on Field-Programmable Gate Arrays, 2017. 217–226
80 Maass S, Min C, Kashyap S, et al. Mosaic: processing a trillion-edge graph on a single machine. In: Proceedings of

the 12th European Conference on Computer Systems, 2017. 527–543
81 Zhou S, Chelmis C, Prasanna V K. High-throughput and energy-efficient graph processing on FPGA. In: Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines, 2016. 103–110
82 Ham T J, Wu L, Sundaram N, et al. Graphicionado: a high-performance and energy-efficient accelerator for graph analytics. In: Proceedings of IEEE International Symposium on Microarchitecture, 2016. 1–13
83 Ozdal M M, Yesil S, Kim T, et al. Energy efficient architecture for graph analytics accelerators. In: Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture, 2016. 166–177
84 Zhuo Y W, Wang C, Zhang M X, et al. GraphQ: scalable PIM-based graph processing. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019. 712–725
85 Deng J, Wu Q, Wu X, et al. Demystifying graph processing frameworks and benchmarks. Sci China Inf Sci, 2020, 63: 229101

Research on the application of graph processing in ATPG

Fubing MAO[1,2,3,4], Da PENG[1,2,3,4], Yu ZHANG[1,2,3,4*], Xiaofei LIAO[1,2,3,4], Xinyu JIANG[1,2,3,4], Yun YANG[1,2,3,4], Hai JIN[1,2,3,4], Jin ZHAO[1,2,3,4], Haikun LIU[1,2,3,4] & Liuzheng WANG[5]

1. *National Engineering Research Center for Big Data Technology and System, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
2. *Services Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
3. *Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
4. *School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
5. *Huawei HiSilicon Co., Ltd., Shenzhen* 518129*, China*
* Corresponding author. E-mail: zhyu@hust.edu.cn

Abstract Automatic test pattern generation (ATPG) is a very important technology in very large-scale integration circuit testing, and its quality directly affects the test cost and overhead. However, existing parallel ATPG approaches generally have problems, such as load imbalance, single parallel strategy, expansion of the storage overhead, and poor data locality. Due to the advantages of high parallelism and high scalability of graph computing, graph processing systems that are fast and efficient and have low storage overhead and high scalability may be significant tools to effectively support ATPG, which will be particularly important for reducing test costs. This paper explores the application of graph processing in combined ATPG. We introduce the graph processing model, which transforms an ATPG algorithm into a graph algorithm; analyze the challenges faced by existing graph processing systems applied to ATPG; and propose a stand-alone graph processing system for ATPG. Then, we discuss the challenges and future research directions of graph processing systems supporting ATPG in terms of optimizing traditional architectures, accelerating emerging hardware, and optimizing emerging storage devices.

Keywords graph processing, VLSI, ATPG, electronic design automation, circuit test