RidgeWalker：FPGA 上的完美流水线化图随机游走

Hongshi Tan[†]、Yao Chen[†]、Xinyu Chen[‡]、Qizhen Zhang[§]、Cheng Chen[††]、Weng-Fai Wong[†]、Bingsheng He[†]新加坡国立大学[†]、香港科技大学（广州）[‡]、多伦多大学[§]、字节跳动[††]hongshi@u.nus.edu、yaochen@nus.edu.sg、xinyuchen@hkust-gz.edu.cn、qz@cs.toronto.edu、chencheng.sg@bytedance.com、wongwf@comp.nus.edu.sg、hebs@comp.nus.edu.sg

arXiv:2601.11057v1 [cs.AR] 16 Jan 2026

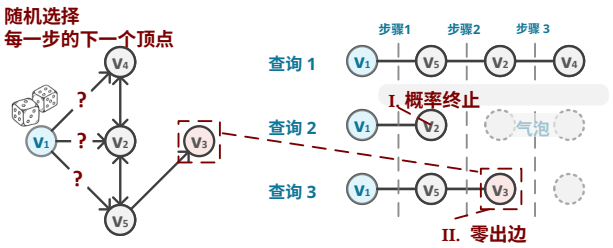
摘要——**图随机游走（GRW）能够高效近似关键的图属性，已广泛应用于众多领域。然而，由于强数据依赖、不规则的内存访问模式以及不均衡的执行行为，GRW 工作负载一直难以加速。尽管近期研究开始探索用于 GRW 的基于现场可编程门阵列的加速器，但由于流水线化效率低下和静态调度，现有方案远未充分发挥硬件潜力。本文提出 RidgeWalker，这是一种面向数据中心 FPGA 设计的高性能 GRW 加速器。RidgeWalker 背后的关键洞见是，GRW 的马尔可夫性质允许将其分解为无状态的细粒度任务，这些任务可以乱序执行而不会影响正确性。基于这一洞见，RidgeWalker 引入了一种异步流水线架构，并配备了以排队论为基础的反馈驱动调度器。该设计实现了完美流水线化和自适应负载均衡。我们在 FPGA 上实现了 RidgeWalker 原型，并在一系列 GRW 算法和真实世界图数据集上评估了其性能。实验结果表明，与最先进的 FPGA 方案相比，RidgeWalker 平均实现了 $7.0\times$ 的加速比；与 GPU 方案相比，平均实现了 $8.1\times$ 的加速比，峰值加速比分别达到 $71.0\times$ 和 $22.9\times$。源代码已公开发布于 *https://github.com/Xtra-Computing/RidgeWalker*。**

I. 引言

现代数据科学和网络分析应用越来越依赖图随机游走（GRW）来高效探索和刻画复杂的网络结构。GRW算法基于马尔可夫过程 [1],，仅根据当前顶点以概率方式选择邻居，从而遍历图，如图1a所示。这种采样策略无需详尽列举，便可近似获得图的关键特征，例如中心性、影响力传播和社区结构。

穷举遍历 [2]–[5]。因此，GRW 已被用作数据库 [6], [7], 、机器学习 [8]–[11], 和基于图的推理 [12]–[14] 的核心构建模块。

GRW已成为许多图学习工作负载中的主要性能瓶颈，通常占总执行时间的50%以上 [15]。这推动了大量研究，致力于在CPU和GPU上开发高性能GRW解决方案 [15]–[20]。然而，GPU等通用架构是为具有大规模规则并行性的工作负载而设计的，因此并不适合具有不规则计算和高度随机内存访问模式的GRW [21]。因此，即使是最先进的基于GPU的GRW [15] 也仅利用了总随机访问



(a) 图上的 GRW 查询遍历。 **(b) 并发 GRW 查询执行。**

图 1：GRW 中固有随机性和不平衡遍历的示意图。

内存带宽的0.9%（见第VIII-C节），这暴露了GRW工作负载与传统硬件之间的根本不匹配。

这种不匹配凸显了针对 GRW 的独特特性设计领域专用架构的必要性，以提高带宽效率和性能。凭借架构可重构性以及对高带宽内存 (HBM)的访问能力，现代 FPGA 越来越受到关注 [18], [22], [23] ，可用于构建定制的 GRW 加速器，并适应快速演进的图随机游走应用。然而，尽管潜力巨大，现有的基于 FPGA 的 GRW 加速器仍未能充分发挥硬件的能力。这一性能差距源于 GRW 工作负载的两个固有特性，它们使高性能加速器的设计尤为困难。

首先，**GRW具有强烈的数据依赖性，且随机内存访问难以预测**。游走的每一步都需要随机访问，以获取当前顶点的邻居；只有完成这次访问后，才能执行下一步（如图1a所示）。与传统图处理加速器类似， [24], [25], 现有的 GRW 设计 [22] 会根据顶点的访问频率，将频繁访问的顶点缓存到片上。然而，由于邻居选择具有概率性，这类策略对于 GRW 基本无效。如图1a所示，即使游走从同一顶点出发，也会迅速沿不同路径分散 v_1 ，局部性很低，因此缓存复用有限。我们对 FastRW 的性能分析 [22] 表明，一旦图超出片上缓存容量，随机访问内存带宽利用率最多会下降 42%。

其次，**并发 GRW 查询执行会导致显著的工作负载不均衡**。由于概率终止以及真实世界图的结构稀疏性，各查询的长度差异很大。例如，图1b中的顶点 v_3没有出边，因此任何到达该顶点的游走都会立即终止。这种情况会导致各查询的执行时间难以预测。现有 GRW架构 [18], [22], [23]依赖静态调度，无法适应运行时的不均衡，从而产生流水线气泡，并降低计算流水线的利用率。我们的评估表明，在这种不均衡情况下，有效的随机访问内存带宽利用率可能低于硬件理论峰值的 2.3%（见第三节）。

我们将现有设计的性能差距归因于缺乏完全流水线化的执行 [26], ，而这要求（a）主动解析数据依赖，以及（b）消除流水线气泡。我们的关键洞见是，GRW 的马尔可夫特性将步骤间的数据依赖限制在当前顶点上，从而使查询能够被分解为无状态、最小化的任务，以实现细粒度执行，同时不损害统计正确性。这一特性促使我们设计出一种 GRW加速器架构，支持乱序、异步查询执行，以隐藏内存延迟，并动态适应工作负载不均衡。

本文提出了 RidgeWalker，一种高效的基于现场可编程门阵列的 GRW 加速器。该加速器利用 GRW 的马尔可夫性质，实现细粒度任务分解，并达到完全流水线化的执行。首先，RidgeWalker 通过异步流水线架构实现乱序 GRW 查询执行。通过交错处理来自不同查询的任务并独立执行这些任务，查询可在运行时动态调度并重新分配到各条流水线，而不再受顺序输入顺序或对处理流水线的亲和性限制。这种灵活的执行模型能够有效隐藏随机内存访问的延迟。其次，RidgeWalker 集成了反馈驱动调度器，可自适应地解决工作负载不均衡和流水线气泡问题。基于排队论，该调度器为零流水线气泡提供形式化保证，即使在高度不均衡的 GRW工作负载下，也能维持连续的数据流。总而言之，本文的贡献如下：

•我们提出了 RidgeWalker，这是首个实现完全流水线化执行的 GRW 加速器，其架构设计与 GRW 算法的马尔可夫结构相契合。

- 我们提出了一种异步 GRW 加速器架构，能够高效地对分解后的无状态 GRW 任务进行流水线化处理，从而实现细粒度并发，并近乎最优地利用随机访问带宽。
- 我们提出了一种基于排队论的反馈驱动调度器，可在不规则工作负载和高并行性条件下，为消除流水线气泡并持续保持高利用率提供形式化保证。
- 我们在多种现场可编程门阵列上实现了 RidgeWalker，并展示了它在多种 GRW 算法和数据集上的通用性与性能；与以往的现场可编程门阵列和图形处理器解决方案相比，其加速比分别高达 71.0× 和 22.9×。

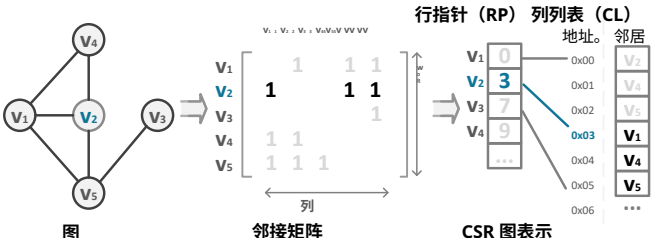


图 2：CSR 格式图表示示例。

算法 II.1： 通用 GRW 算法

输入： $\mathcal{G}$：给定的图，$\mathbb{Q}$：一组输入查询。
输出： *res*：遍历顶点路径。

```
结果 = ∅;
遍历 Q ∈ ℚ，执行
    v_curr = Q.v_start，路径 = ∅;
    loop
        /* 访问压缩稀疏行的行指针； */
        {地址，度} = 行访问(v_curr， 𝒢);
        /* 特定应用采样； */
        索引 = 采样(地址， 度 𝒢);
        /* 访问 CSR 的列列表； */
        v_curr = 列访问(地址， 索引 𝒢);
        路径.推送(v_curr);
        如果 (Q.是结束())则
            结果。压入(路径);
            中断;
返回结果;
```

II. 背景

A. CSR图表示

图 2 展示了一个采用 压缩稀疏行 （压缩稀疏行）格式编码的示例图，这是 GRW 工作负载中最常用的邻接表示。压缩稀疏行使用两个数组：行指针数组（RP）和列列表数组（CL）。每个 RP$[i]$ 条目都指定了顶点v_i的邻居列表在 CL 中的起始偏移量；CL 存储邻接矩阵中非零元素的列索引。例如，RP$[2]$ $=$ 3表示 v_2 的邻居从 CL 地址 0x03 开始存储。将每个顶点的邻居连续存储，可实现 O(1) 次索引查找，因此非常适用于 GRW 中的随机采样，并得到广泛采用 [15]–[20]。

B. GRW执行流程

算法 II.1 概述了一般 GRW 算法的执行流程。输入为一个 CSR 图 $\mathcal{G}$，应用程序发起一组随机游走查询 $\mathbb{Q}$ ，并指定起始顶点。查询从 v_{curr} $=$ $Q.v_{start}$ 开始，每次前进一步。对于每一步，算法首先从 CSR 图的行指针数组中读取 v_{curr} 的度数以及指向其邻居列表的指针（第 5 行）。随后，特定应用采样函数从该列表中选取一个邻居索引（第 6 行）。利用选定的索引，算法访问列列表，获取下一个要访问的顶点

。当游走达到最大长度或遇到没有出边的顶点时，查询便终止。

GRW 天然难以实现完美流水线化，因为其执行过程无法组织成完美嵌套循环 [27], ，这是因为内层循环的迭代次数无法预先确定。这种不可预测性使得依赖静态调度的传统流水线技术（如带数据通路的有限状态机 (FSMD) [28], 、基于 SDC 的模调度 [29],及类似方法）无法直接应用，也难以实现完美流水线化和最优带宽利用率。

III. 动机与设计原则

A. 图随机游走专用加速器的必要性

尽管已有大量研究聚焦于通用图处理加速器 [24], [25], [30]–[38], ，但这些设计从根本上并不适合 GRW 的需求。通用图处理通常需要扫描某个顶点的所有邻居，以进行聚合或消息传播。因此，人们提出了更新合并（例如 ACTS [30], 、Swift [32]）和顺序缓冲（例如 Graph-Pulse [35]）等优化方法，以高效利用片上存储器。然而，GRW 在结构上有所不同：它们在图中遍历时，每一步只采样一个随机选定的邻居，且不进行聚合。这种稀疏的访问模式使此类缓冲和聚合技术无法奏效。因此，亟需专门设计能够适应 GRW 随机且数据依赖行为的架构。

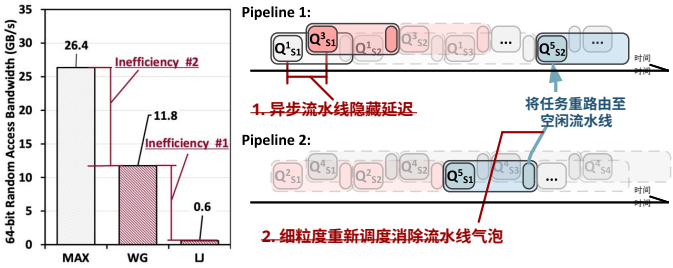
B. 现有 *GRW* 加速器的低效之处

尽管此前已有多项研究探索基于现场可编程门阵列的 GRW 加速，但当前设计远未发挥硬件的潜力。为探究 GRW 加速器的设计挑战，我们对 FastRW [22]和 LightRW [18]进行了深入的性能分析。我们测量其带宽利用率，定义为 $B_{\text{measured}}/B_{\text{peak}}$，其中 B_{measured} 是 GRW 遍历期间观测到的有效带宽，通过将遍历边的总内存占用量除以总执行时间计算得出。 B_{peak} 表示动态随机存取存储器或高带宽内存提供的理论峰值带宽，按照 Asifuzzaman 等人 [39] 的方法进行估算。由于每个 GRW 步骤通常都会触发一次动态随机存取存储器行缓冲区未命中，我们据此计算峰值 64 位随机访问带宽 B_{peak}，其粒度与 GRW 中顶点级随机访问的粒度一致：

$$B_{\text{peak}} = \frac{f_{\text{mem}}}{t_{\text{RRD}}} \times N_{\text{chn}} \times \frac{64\text{-bit}}{8}, \tag{1}$$

其中， f_{mem} 为内存运行频率， t_{RRD} 为行间延迟， N_{chn} 表示可用内存通道的数量。我们总结出两个关键观察：

观察 #1：由于 GRW 的数据依赖，内存访问延迟本质上难以隐藏。图 3a 显示，在较小的 WG图 上，FastRW [22] 可维持 11.8 GB/s 的有效带宽，此时行指针数组完全适合存放在片上SRAM 中。然而，在较大的 LJ图 上，性能骤降至 0.6 GB/s，仅为峰值的 2.3%。



（a）自底向上分析。 **（b）提出的完全流水线化方案。**

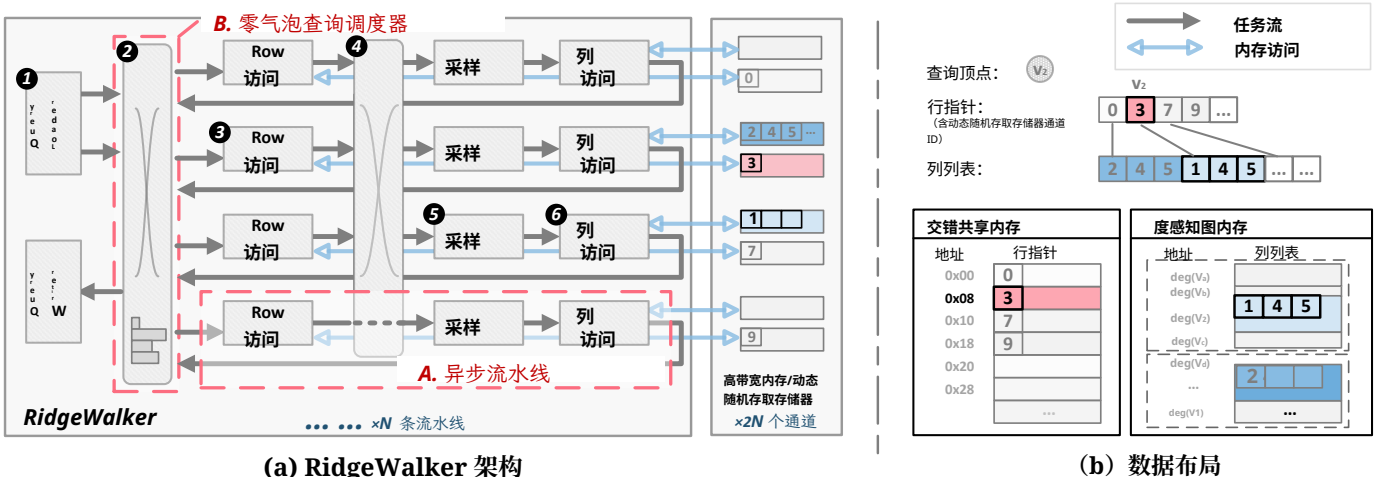
图 3：（a）SOTA 现场可编程门阵列加速器 Fas- tRW [22], 的带宽分析，凸显了利用不足。（b）在两条流水线上完美流水线化的并行 GRW 执行， Q_{sy}^{x} 表示查询 x遍历的第 y步。

这一降幅揭示了一个根本性挑战：GRW 施加了严格的逐步数据依赖，每一步都必须完成一次随机内存访问，以确定下一个顶点，然后才能继续。这些访问不仅是随机的，而且按顺序串联，使得跨步骤进行预取或并行化极其困难。因此，仅优化内存层次结构是不够的；要隐藏延迟并实现高性能，关键在于解决 *GRW* 的步间依赖。

观察 #2：静态调度无法解决 GRW 中的工作负载不均衡问题。图 3a 将 FastRW 的实测带宽与根据公式（1）推导出的理论峰值 MAX 进行了比较。即使行指针数组完全适合片上 RAM，FastRW 也只能达到峰值的 45%。性能差距并非内存问题，而是由查询调度造成的。各个查询的完成时间不同，因此固定调度在等待最慢查询时会不断累积空闲周期。LightRW [18] 通过在环形缓冲区中批量处理查询来改善局部性，但仍按预定顺序发出每一步；当一次游走提前终止时，为其预留的槽位便会空置。我们对 LightRW 的分析发现，气泡率最高可达 37%，证实静态调度无法应对 GRW 高度不均衡的运行时行为。要弥补这一差距，调度器必须能够动态调整，将就绪任务转发至空闲的处理流水线，从而使所有硬件资源保持繁忙。

C. *RidgeWalker*：迈向完美流水线化的 *GRW*

我们的分析表明，要高效加速 GRW，就必须重新思考算法与架构，以打破数据依赖并动态平衡工作负载。我们的核心洞见基于 GRW 的马尔可夫性质：随机游走的下一步仅取决于当前顶点，与此前的历史无关。这一性质使我们能够将每个 *GRW* 查询分解为无状态的细粒度任务，这些任务仅涉及局部化内存访问与计算。 这些任务本质上彼此独立，无需维护全局游走状态，也不需要任务间同步即可执行，从而使架构能够发出大量未完成的内存请求，并以更细的粒度平衡工作负载。基于这一洞见，我们提出——



(a) RidgeWalker 架构 **（b）数据布局**

图 4：RidgeWalker 架构概览。

了 RidgeWalker，这是一种 GRW 加速器，通过两项协同设计创新实现了完美流水线化执行：

乱序查询执行。每个 GRW 遍历跳还会进一步分解为多个相互独立的无状态任务，例如内存访问和采样；一旦所需的顶点数据可用，这些任务便可立即发出。不同查询的任务可以自由交错执行，从而利用其他查询中的就绪任务掩盖某个查询的内存停顿（例如，Q^1_{s1}、 Q^1_{s2} 和 Q^3_{s1} 在图 3b 中共享流水线 1），使内存延迟与计算完全重叠。

工作负载自适应调度。 为消除并发 GRW 查询执行中的流水线气泡，RidgeWalker 以周期级粒度持续监控流水线利用率，并通过反馈驱动调度动态地将就绪任务重定向至空闲模块。例如，在图 3b 中，当 Q^5_{s1} 在流水线 2 中完成且流水线 1 变为空闲时，调度器会立即分派 Q^5_{s2} 来填补空档。这种自适应调度可消除流水线气泡，即使面对高度负载不均衡的 GRW，也能维持峰值吞吐量。

IV. RIDGEWALKER 概述

A. 架构概述

图4展示了 RidgeWalker 的架构。RidgeWalker 是一种新型 GRW 加速器，能够实现完美流水线化的执行。RidgeWalker 采用了两个紧密集成的组件：

A) 异步流水线（§V）： 为了支持 GRW 查询的乱序执行，RidgeWalker 在可扩展架构中引入了异步流水线设计。每条异步流水线都通过三个关键模块独立处理 GRW 步骤的各个阶段：行访问、采样和列访问。行访问和列访问模块分别分配了独立的 HBM 通道，从而避免通道间仲裁和争用，确保一个阶段发出的随机访问不会干扰另一阶段发出的访问。

在每条流水线中，RidgeWalker 集成了一个异步内存访问引擎（见第 V-B 节），可提供多达 128 个未完成的非阻塞请求。该引擎采用全流水线化设计，启动间隔为一个周期，使其能够充分利用 HBM 控制器的未完成请求容量，并使内存吞吐量与流水线的处理速率相匹配。分配给每个访问引擎的 HBM 通道数量，依据异步流水线对随机访问带宽的需求来确定。例如，单个 HBM2 通道每秒可持续处理约 2.84 亿次 64 位随机事务。因此，以约 300 MHz 运行的内存访问引擎与单个 HBM 通道的容量十分匹配，能够充分利用其可用的随机访问带宽。

B) 零气泡查询调度器（§VI）： RidgeWalker 实例化了多条异步流水线，以便在执行并行 GRW 查询时充分利用可用的随机访问内存带宽。为支持数据层面的这种并行性，CSR 图被随机划分，并分布到所有 HBM 通道上。与广度优先搜索或 PageRank 不同，后两者的遍历与分区边界紧密相关，往往会导致严重的负载不均衡；GRW 的访问分布则均匀得多。关于 GRW 混合特性和相遇时间的先前分析 [40], [41] 表明，任何不均衡都只会持续很短的步骤窗口；之后，各分区的访问概率会趋于均匀（对于百万规模的图，通常只需几十个步骤）。

为应对这些短暂的波动，RidgeWalker 允许在每一跳灵活地重新分配每个游走体。多个并行流水线异步运行，而采用蝴蝶互连网络实现的任务路由器会根据任务需要访问的顶点，将每个任务路由到正确的内存通道。零气泡调度器持续监控流水线状态，并立即用就绪任务填满空闲时隙，确保突发的临时负载不会降低吞吐量。灵活的重新调度与高吞吐量路由相结合，使 RidgeWalker 即使在完全随机的 GRW 访问模式下也能保持线速执行。

B. 查询执行流程

图 4a 展示了 GRW 查询如何在 RidgeWalker 中执行单步操作，图 4b 则展示了相应的图内存布局。为匹配 GRW 访问模式，CSR 图被映射并分布到各个高带宽内存通道中：行指针数组被分区并分配到行访问通道，而邻居列表则以轮询方式在列访问通道之间打乱分布，以降低发生存储体冲突的可能性。每个行指针条目都编码了目标通道 ID 以及对应邻居列表的起始地址，从而便于直接访问。

对于从顶点 V_2开始的查询，查询加载器 (❶)从主机内存中获取该顶点；调度器 (❷)将其分配给流水线 2；行访问 模块 (❸) 获取 V_2 的行指针。如果邻居列表位于不同的通道中，任务路由器 (❹)会相应地重定向该任务。采样模块 (❺)根据可配置分布选择一个邻居，而列访问模块 (❻)则获取该邻居并确定查询是否终止。各模块通过 AXI-Stream 协议中的浅 FIFO 进行通信，从而实现基于背压的流量控制。查询终止后，系统会根据查询索引收集遍历的顶点列表。此过程以流式窗口方式运行。当累积路径长度达到预定义的写入粒度时，相应路径将依次写回全局内存或主机内存，供下游应用使用，且不会中断 GRW 的执行。

V. 乱序并发查询执行

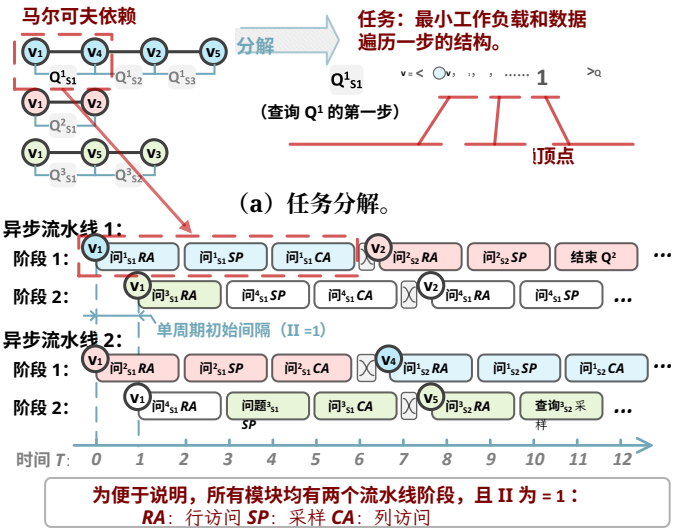
本节介绍所提出的异步流水线如何实现乱序查询执行并解决数据依赖，以及异步内存访问引擎如何摊销内存延迟，从而在 GRW 高度随机的访问模式下最大化带宽利用率。

A. 将查询分解为无状态任务

不同于传统的数据流架构 [18], [22], [42],（在这类架构中，执行严格受数据可用性支配，且每个模块都维护本地状态以确保顺序控制），RidgeWalker 通过基于马尔可夫的任务分解，以及随后进行的无状态乱序流水线执行，消除了这一限制。每个分解后的任务都能独立流经流水线，无需额外的控制逻辑或全局同步。任务会被标记为一个唯一的

用于结果跟踪的查询索引，使系统能够正确地将采样的顶点与对应的查询关联起来。每个模块仅根据输入任务是否可用来执行，并且与其他模块的状态完全解耦。因此，任务执行和重新调度不会发生阻塞或相互干扰，从而支持完全乱序的查询处理。

图 5a 展示了 RidgeWalker 如何将一个 GRW 查询分解为最小且相互独立的任务。每个任务都表示为一个元组 $Q^y_{sx} = \langle v_{\text{last}},\ \text{ID}_y,\ x,\ \ldots \rangle$，其中 v_{last} 表示



(b) 乱序查询执行。

图 5：基于马尔可夫的任务分解，以及跨两个流水线的乱序查询执行示意图。

最近访问的顶点（对于 Node2Vec 这类高阶游走，则表示最近访问的两个顶点），ID_y 唯一标识该查询，而 x记录当前跳数。此元组表示可容纳在单个流水线字中。每个模块在一个周期内读取该字、执行计算、更新该字，并将其转发给下一个模块。所有模块的设计目标都是每个周期处理一个任务，从而维持全流水线化的吞吐量。

图 5b 展示了分解后的步骤如何在多个流水线中乱序执行。每个流水线都由行访问（RA）、采样（SP）和列访问（CA）模块组成；为便于说明，每个模块简化为两个流水线阶段，启动间隔为一个周期。得益于任务间的独立性，这些任务可根据运行时的可用情况动态路由至不同流水线，从而灵活、高效地利用所有处理资源。例如，在 $T = 6$ 时，流水线 1 已完成 Q^1_{s1}，其后续步骤 Q^1_{s2} 随即被分派至流水线 2，从而消除流水线间阻塞。同样，Q^3 的第一跳在流水线 1 中开始，但后续步骤会转至流水线 2，以访问映射到其关联内存中的图数据。通过将流水线模块完全解耦，并借助乱序执行模型移除集中式控制，RidgeWalker 得以无缝地将模块级访问与计算重叠执行。

B. 用于并发执行的异步内存访问

异步访问引擎是支撑行访问和列访问模块的核心微架构组件。图 6 展示了其架构。传入的任务首先进入请求代理，由其提取目标地址及相关元数据（例如顶点索引和查询 ID）。顶点索引会被转换为物理地址，并转发至内存引擎，后者以内存的最小粒度发出动态随机存取存储器访问请求。

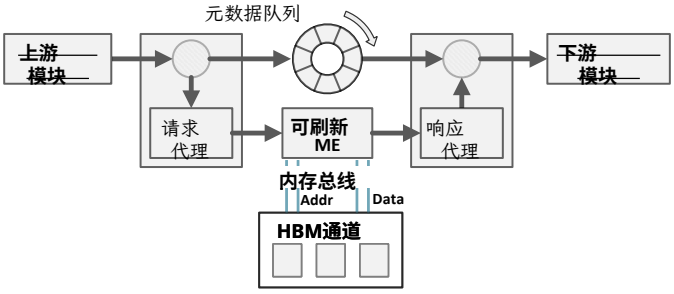


图 6：异步内存访问引擎的架构。

元数据绕过数据路径，单独入队，以便与返回的数据重新合并。

当输入数据不可用时，传统硬件流水线会停顿，导致串行化，并且无法有效隐藏延迟。为避免这种情况，内存引擎独立于输入就绪信号运行，使访问请求能够继续执行并顺畅完成，不受延迟影响。每个内存请求都通过 AXI 协议发出，并附带唯一的事务 ID。相关元数据会被加入基于块RAM的元数据队列，其容量足以覆盖往返延迟（最多 512 个条目，足以满足 320 MHz 下 100 个周期的需求）。由于 AXI 协议保证每个事务 ID 的响应按序返回，我们的内存引擎包含一个片上缓冲区，支持最多 64 个事务 ID，以重组乱序返回的数据。随后，响应代理 会将数据和元数据重新组装成完整任务，供下游模块使用。该设计消除了流水线停顿，充分隐藏了内存延迟，并支持高吞吐量、细粒度的随机访问，这对 GRW 至关重要。

C. 并行流水线上的动态查询重新分配

为了有效地将 GRW 执行扩展到大量并行流水线和内存通道，RidgeWalker 必须支持查询的逐跳动态重新分配。RidgeWalker 架构中的重新分配灵活性源于两个相互正交的维度。

首先，我们基于马尔可夫的逐跳任务分解确保每个 GRW 跳都表示为一个独立的无状态任务。由于跳与跳之间不携带历史状态或依赖链，任何分解后的任务都可以由任意流水线执行，而不会影响正确性。这为完全灵活的逐跳重新分配奠定了基础。其次，在 FPGA 结构上实现动态任务路由的硬件开销极低。每个分解后的任务都很紧凑，不超过 512 位，并且可通过 AXI-Stream 接口在单个周期内传输。这些接口只需少量资源；即使是用于流量控制的握手 FIFO，也可以用一个可配置逻辑块（32 个条目）实现，这已足以维持模块间的流水线化。这种紧凑且自包含的任务格式使得仅凭轻量级片上互连和简单的调度逻辑，就能在众多并行流水线之间路由和重新分配任务。

鉴于这种灵活性，关键设计目标是通过考虑两个因素来保持执行均衡：（1）每个任务都必须被路由到存储其当前顶点邻接表的正确内存通道；（2）任何流水线都不应空闲，从而最大限度地提高吞吐量。为此，RidgeWalker 集成了两种利用任务独立性的架构机制。任务路由器采用蝴蝶互连网络实现，通过将每个任务定向到存储所需邻接表的内存通道，执行数据感知路由。与此同时，零气泡调度器 持续跟踪流水线的可用情况，并立即将就绪任务发射到新释放的流水线槽位中。由于任务不包含可变状态，因此每个周期都可以重新分配任务，无需回滚、协调或同步。因此，我们的架构能够无缝吸收短暂的工作负载变化，并防止气泡形成。

通过结合无状态任务分解、数据感知路由和运行时任务分派，RidgeWalker 能够在所有流水线和内存通道上保持高利用率，同时提供可扩展、完全流水线化的 GRW 执行所需的灵活性。

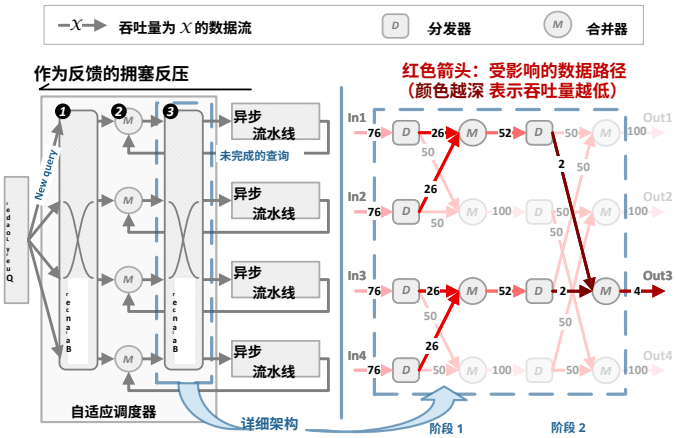
VI. 并行 GRW 的零气泡调度

图7展示了零气泡调度器的架构。它根据实时执行状态，动态地将 GRW 任务分派到流水线。我们的设计以排队论模型为基础，可确保异步执行时达到完全利用率，并保证任务执行过程中不会产生流水线气泡。

A. 问题表述

硬件视图： 如图7所示，GRW 执行由 N 异步流水线构成，这些流水线通过流式 FIFO 和一个 N-到-N 平衡器连接。每个 GRW 查询都被分解为一系列无状态任务，但每个查询的执行时间取决于数据：内循环迭代次数（邻居遍历、拒绝重试、提前终止等）事先无法确定。因此，不同流水线完成查询的时间各不相同，反压会沿 FIFO 和平衡器传播。因此，核心挑战是一个硬件调度问题：每个周期中，调度器必须仅依据实时可用性信号（空/满）来决定向哪些流水线提供任务，从而确保：i）只要存在工作，就不会有流水线处于空闲状态（零气泡）；ii）在发生拥塞时，不会总是优先选择某一部分流水线（公平性）。

排队模型： 我们将调度器建模为批量服务队列，具体而言是一个 $M/M/1[N]$ 系统 [43], ，以反映硬件约束：调度器可以同时向多个流水线发出任务。在不失一般性的前提下，我们假设输入任务以速率 λ 按泊松过程到达，从而刻画上游模块随机注入 GRW 任务的过程。任务服务时间建模为速率为 μ的指数分布，以反映 GRW 工作单元完成时间的随机性及其对图结构的依赖性。单个“服务器”对应调度器/均衡器逻辑，每个决策时刻最多可分派 N 个任务



(a) 零气泡调度器。 **(b) N 对 N 平衡器。**

图 7：零气泡查询调度器架构，以及在 $N = 4$ 条流水线上平衡工作负载的示例。

（即一个周期），与 N 条并行流水线相匹配；因此，我们排队模型中的最大批量大小为 N。

反压与观测延迟：在硬件中，调度器无法观测流水线的即时空闲状态；它只能通过 FIFO 占用量和反压信号来观测系统，而这些信号会受到互连和流水线深度的影响而产生延迟。我们用延迟 C 个周期的观测来描述这一效应：在周期 t，调度决策依据反映周期 $t - C$ 工作负载状态的可用性信号作出。这种延迟反馈与随机服务时间共同导致了朴素静态调度中的气泡。我们的目标是设计一种可硬件实现的策略，在高负载下保持稳定（即不造成队列无限增长），并且只要有任务存在，即使存在 C 个周期的延迟，也能使 N 条流水线保持满负载运行。

B. 零气泡调度的缓冲需求

根据图7中的硬件执行模型，调度器通过FIFO反压观察流水线可用性，其反馈延迟是有限的（最长为 C 个周期）。在这种延迟观测下，即使上游存在任务，不足的缓冲也可能暂时使就绪流水线得不到任务，从而产生气泡。因此，我们应用Lu 等人 [44]的排队论结果（该结果在定理 VI.1中形式化），其刻画了分发器（服务器）与下游服务节点之间为在延迟反馈下维持稳态所需的最小队列深度。具体而言，配置深度至少为 $N + N\mu C$的队列，可确保调度器尽管存在 C个周期的延迟以及随机服务时间变化，仍能持续提供任务。因此，只要系统中存在待处理任务，全部 N 条流水线就能保持充分利用，实现零气泡调度并维持峰值吞吐量。

定理 VI.1 (最小缓冲区队列深度)。 考虑一个系统，其中有 N 个独立服务器处理任务，每台服务器每个周期最多可处理 μ 个任务。调度器从队列中调度任务，并接收有关服务器可用性的反馈，最大延迟为 C_{max} 时间。为确保所有服务器始终满负荷运行且不空闲，最小队列深度 D 必须满足：

$$D = N + O(\mu C_{max} N). \tag{2}$$

C. 从排队保证到硬件调度器架构

定理 VI.1给出了由延迟反馈导致的气泡在原则上得以消除的缓冲条件。接下来，我们将这一条件转化为可在硬件中实现的调度器：一种全流水线化、无分支的分发结构，利用FIFO满/空信号路由任务，并借助推导出的队列深度吸收 N 条异步流水线之间的工作负载不均衡。

如图7a所示，零气泡调度器由三个核心功能模块组成：❶、❷和❸。这些模块共同构成一条流水线化的调度数据通路，旨在实现高吞吐量和无气泡执行。模块❶充当初始任务均衡器，根据流水线的可用情况，将查询加载器新注入的查询自适应地分配到可用的调度路径。它确保输入任务的调度不会因批量查询加载而停滞。模块❷充当任务合并器，将模块❶新调度的查询与从下游流水线返回的未完成查询合并。该模块优先处理在途未完成查询，使其能够立即重新导入调度流水线，而无需额外等待。最后，模块❸充当具备反压感知能力的分发器，根据信号实时指示的可用情况，将已就绪任务路由至相应的处理流水线。这种多级协调使调度器能够动态均衡负载，并在执行槽释放时将其填满，从而消除气泡并维持最高的流水线利用率。

1） N到N 任务均衡器设计：传统的任务调度方法（包括完全公平调度器 [45]–[47],）在跨 N 个处理器调度 N 个任务时，需要进行原子调度，才能在 $O(N\log N)$时间复杂度内分配所有任务。调度器必须通过轮询所有处理器的状态，在一个集中式、不可分割的操作中分配任务，从而选定并提交全局调度决策。每次调度操作中，每个任务都涉及 $O(\log N)$ 的复杂度，并会在处理器之间引入强同步依赖。

相比之下，我们的任务均衡器使用轻量级分发器和合并器模块，将调度过程分解为任务之间相互独立的成对比较，每个模块的运行时间为 $O(1)$。如图7b所示，这些模块还通过基于蝶形网络的多级拓扑进行扩展。在每个阶段，两任务比较的结果都会向前传递，并参与下一阶段的仲裁，以解决分布不均衡问题。每个模块都是全流水线化的，可实现线速调度，并持续适应运行时工作负载分布。

图7b展示了任务均衡器如何缓解下游负载不均衡。假设第三个输出通道的速率被限制为 4 pkt/s，而其他通道均能维持 100 pkt/s。在

算法 VI.1： 平衡任务分派。

输入： *in*：输入任务流。
输出： *out_1*：第一个输出任务流；
out_2：第二个输出任务流。

```
1  最后一次选择 = 0;
2  do
3      如果 任务 =非阻塞读取() 则
           scode =构建_scode(上一次选择,
4                            输出_1。已满(),
                             输出_2。已满());
5          开关 scode 执行
6              情况 0b001 执行
                   // 两者都有空间；选择
                      上次未服务的项交替
                      （选择输出_1）。
7              情况 0b111 时执行
                   // 两者都已满；阻塞上次未服务的项
                      （输出_1）以保证公平性。
8              情况 0b101 , 0b100 执行
                   // 只有一个通道可以接受 (out_2
                      full)；路由到 out_1 以避免
                      停滞。
9                  输出_1。阻塞写入(任务);
10                 最后一个选择 = 0;
11                 跳出;
12             否则执行
13                 输出_2。阻塞写入(任务);
14                 最后一个选择 = 1;
15                 跳出;
16 当True;
```

算法 VI.2： 平衡任务合并。

输入： *in_1*：第一个输入任务流；
in_2：第二个输入任务流。
输出： *out*：输出流。

```
1  上一次选择 = 0;
2  do
       scode =构建_scode(上一次选择,
3                        输入_1。为空(),
                         输入_2。为空());
4      开关 scode 执行
5          情况 0b111 , 0b110 执行
               // 两个输入均为空。
6              跳出;
7          情况 0b101 , 0b100 执行
               // 只有一个输入包含有效数据；
                  直接将其转发到输出端
                  （选择 in_1）。
8          情况 0b001 时
               // 两个输入均有效；交替
                  根据上次未服务的情况，以避免
                  在拥塞时发生饥饿（选择
                  在_1）。
9              任务 =在_1。非阻塞读取();
10             输出。阻塞写入(任务);
11             最后一次选择 = 0;
12             中断;
13         否则执行
14             任务 =输入_2。非阻塞读取();
15             输出。阻塞写入(任务);
16             最后选择 = 1;
17             跳出;
18 当真;
```

第二阶段，每个向慢速通道传输流量的分发器都会接收一条快速路径和一条慢速路径的流量，平均速率为 $(100+4)/2 = 52$ pkt/s。这种不均衡在第一阶段进一步得到平均化，使四个输入的速率均达到 76 pkt/s。因此，即使单个下游通道受到限速，多级路由网络也能将局部拥塞分散到上游，并使前面的阶段保持均匀负载。

2) 任务分发器：算法 VI.1 详细说明了分发器如何将任务从单个输入流路由到两个独立的输出通道，同时遵循反压并保持公平性。分发器维护一位状态 last_selection，并在第 1 行将其初始化，以记录最近一次服务的是哪个输出。在每次迭代中（第 2 行），它首先尝试以非阻塞方式读取输入（第 3 行）。如果没有可用任务，分发器便跳过本次迭代，不会使上游停滞。获取任务后，分发器通过 build_scode() 函数（第 4 行）构造紧凑的三位调度码*scode*，将上次选择与*out 1*和*out2*的满载状态打包其中。具体而言，last_selection 占据最低有效位（LSB），而 *out 2* 的满载标志则置于最高有效位（MSB）。

在最坏情况拥塞下保证均衡。这种编码可以通过 switch（第 5–15 行）进行简单解码，而无需复杂的运行时控制分支。在上一个任务被发送到*out 2*的情况下，如果两个输出均可用，分发器会在第 6 行选择上次未服务的通道，以交替提供服务并平衡负载。如果两个输出均已满载，它会在上次未服务的通道上阻塞，避免一侧持续遭到抢占，从而在最坏情况拥塞下保证公平性（第 7 行）。如果只有一个输出可以接收新数据，任务就会直接路由到该可用通道（例如第 8 行）。每次阻塞式_写入后，分发器都会更新 last_selection，以反映所选输出（第 13–14 行），确保后续迭代中持续交替。总体而言，分发器是全流水线化的，启动间隔为一个周期，固定延迟为两个周期。

3) 任务合并器：算法 VI.2 展示了合并器中实现的算法，该算法将两个输入任务流合并为一个输出流，同时在反压情况下保持服务均衡。合并器使用一位状态 last_selection 跟踪最近选中的输入（第 1 行）。在每次迭代中（第 2 行），它

构造一个三位调度码*scode*（第 3 行），其中 last_selection 占据最低有效位，*in 1* 和*in 2* 的空标志分别存储在接下来的两位中。**保证最坏情况拥塞下的公平性**。调度策略如下：如果恰好一个输入包含有效数据，则直接将该输入转发到输出，以最大化吞吐量（例如第 7 行和默认情况，无论上次选择的是哪个输入）。当两个输入都有效时，合并器根据 last_selection 选择上次未服务的流，以交替选择输入（第 8 行）。这样可以防止拥塞时单个输入持续占用输出，避免饥饿，并限制另一个流的最坏情况等待延迟。每次成功转发后，都会更新 last_selection，以反映所选输入（第 11 行），从而确保后续迭代持续保持均衡。与分发器类似，合并器采用全流水线化设计，启动间隔为一个周期，固定延迟为两个周期。

D. 实现完美流水线化

现在，我们分析反馈延迟，以应用定理 VI.1，通过调度消除所有流水线气泡。在蝶形拓扑中，每个任务都会经过任务均衡器上的 $\log N$ 分发器 和 $\log N$ 合并器 单元。由于每个单元都实现了全流水线化，且延迟至多为两个周期，因此任务均衡器中的总延迟上限为 $2\log N$ 个周期。再加上从调度器到所选流水线并返回的往返延迟，总调度延迟至多为 $4\log N$ 个周期。鉴于每条流水线都能维持每周期一个 GRW 步骤的理想吞吐量（即 $\mu = 1$），定理 VI.1 指出，调度器与流水线之间所需的最小队列深度为 $D = N + 4N\log N$，才能保证零气泡执行。这相当于每条流水线配备一个深度为 $1 + 4\log N$ 的 FIFO，从而确保不会有流水线因输入饥饿而空闲。

吞吐量与延迟分析。 我们的调度器旨在消除运行时吞吐量开销。它采用全流水线化的调度数据通路，启动间隔为一个周期，确保新注入的查询和重定向查询能够在多个处理流水线之间同时以最大吞吐量进行调度，且不会发生停滞。重定向查询产生的固定延迟很小（例如，对于 16 条流水线为 8 个周期），可被完全隐藏，并通过流水线执行得到摊销。从下游流水线的角度来看，查询始终持续可用，避免因调度器延迟而产生空闲周期。

VII. 适配不同的 GRW

RidgeWalker 采用模块化、可扩展的设计，支持图学习应用中使用的多种 GRW 算法。采样模块通过标准 AXI-Stream 接口进行通信，从而能够灵活集成自定义采样逻辑。每个实例都配有 Thun-deRiNG [48]，这是一种针对现场可编程门阵列优化的高吞吐量随机数生成器。任务输入格式封装了完整的查询会话上下文，从而支持算法特定采样行为。此外，图表示采用基于模板的设计，以支持加权边和扩展元数据。具体而言，行指针条目（RP_{entry}）的大小可在编译时配置，从而可根据别名表等辅助结构进行定制。

表 I：支持的采样算法和 GRW。

GRWs	加权 $\mathcal{G}$?	采样算法	$\lvert RP_{entry}\rvert$
URW [49],PPR [50]	No	均匀采样	64 位
DeepWalk [5]	Yes	别名采样	256 位
Node2Vec [9]	No	拒绝采样	64 位
Node2Vec [18]	Yes	蓄水池采样	128 位
MetaPath [8]	Yes	蓄水池采样	128 位

RidgeWalker 支持无权图和加权图中所有常用的采样算法，如表 I 所总结。URW [49]和 PPR [50]使用均匀采样。DeepWalk [5]采用别名采样 [51], [52]，这要求每个邻居列表维护一个别名表。为此，RP_{entry} 格式扩展为 256 位，用于存储别名表指针及其大小。对于 Node2Vec [9]，RidgeWalker 同时支持拒绝采样和蓄水池采样，分别适用于无权图和加权图。

为配置采样模块，RidgeWalker 通过 PCIe 提供内存映射 AXI4-Lite 控制寄存器，使主机能够设置特定于算法的参数，例如 PPR 中的传送概率 α，或 Node2Vec 中的偏置因子 p 和 q。所有配置字段均可通过轻量级 32 位寄存器写入进行访问。此外，还可以通过 amodebit 选择采样模式（例如加权或无权），从而在不同 GRW 变体之间快速切换，而无需完全重新综合。

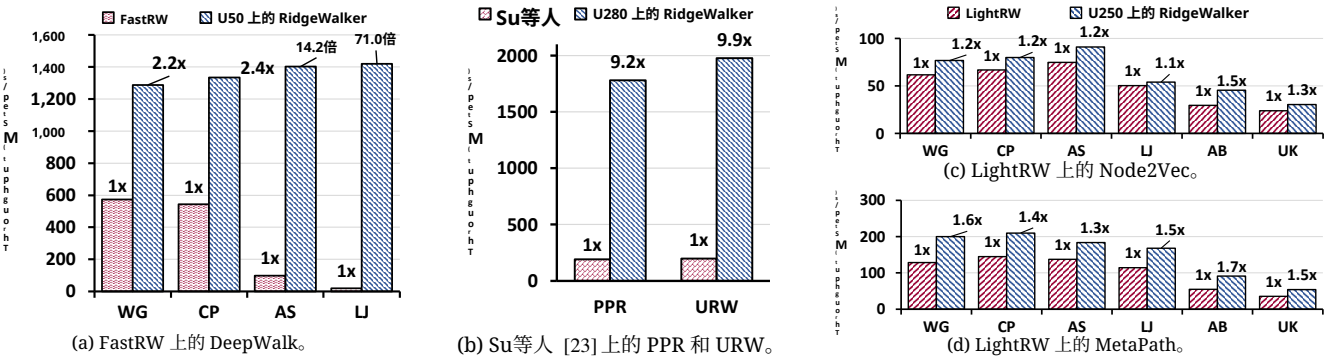
VIII. 评估

本节将全面评估 RidgeWalker，重点关注其异步执行和完美流水线化。我们将其与最先进的解决方案进行比较，并围绕以下目标组织评估：

1. 评估 RidgeWalker 相较于最先进的基于现场可编程门阵列的加速器的性能，并突出其完美流水线化架构设计的优势（§VIII-B）。2. 将 RidgeWalker 与高性能的基于 GPU 的解决方案进行比较，并展示其架构优势，尽管可用带宽较低（§VIII-C）。3. 通过微基准测试和消融研究，定量分析所提出的异步访问引擎和零气泡调度器的有效性，说明我们如何解决观察 #1 和 #2（§VIII-D）。

A. 实验设置

1) 硬件设置：我们使用高级综合（HLS）对 RidgeWalker 设计进行原型实现，并主要在 AMD Alveo U55C FPGA 板卡上评估其性能。性能实验在一台双路 AMD EPYC 7V13 128 核 CPU 服务器上进行。由于每块 U55C FPGA 最多支持 32 个 HBM 内存通道，且每条异步流水线占用两个 HBM 通道，因此我们实例化



(a) FastRW 上的 DeepWalk。 (b) Su等人 [23] 上的 PPR 和 URW。 (c) LightRW 上的 Node2Vec。 (d) LightRW 上的 MetaPath。

图 8：C RidgeWalker 与最先进的基于FPGA的GRW加速器之间的吞吐量比较。

表 II：评估所用的真实世界图数据集。

图	$\|V\|$	$\|E\|$	Size	类别	δ
web-Google (WG) [53]	0.9 M	5.1 M	48 MB	Web	21
cit-Patents (CP) [53]	3.8 M	16.5 米	0.2 GB	引用	26
as-Skitter (AS) [53]	1.7 M	22.2 M	0.2 GB	网络	31
soc-LiveJournal (LJ) [53]	4.9 百万	69.0 M	0.6 GB	社交	28
arabic-2005 (AB) [54]	22.7 M	0。6 B	5.0 GB	Web	133
uk-2005（英国） [54]	39。6 M	0。8 B	6。7 GB	Web	45

$32/2 = 16$ 条异步处理流水线，遵循第 IV 节中的架构设计。相应地，零气泡调度器配置为 16 个输出端口，并连接到异步处理流水线。

2）FPGA 基线：Su等人 [23], 的 FastRW [22]和 LightRW [18] 分别是 PPR、URW、DeepWalk、Node2Vec 和 MetaPath 的先进加速器。为确保评估公平，RidgeWalker 在这些基线所用的同款 FPGA 上重新综合，包括 AMD Alveo U50（FastRW）和 AMD Alveo U250（LightRW）。由于 RidgeWalker 与架构无关，移植仅涉及重新确定时序的内存接口通道数。

3) GPU 基线：对于基于 GPU 的解决方案，我们将 RidgeWalker 与最先进的解决方案 gSampler 进行基准测试，因为 gSampler 在所有评估的应用中都能提供最佳加速性能。实验在一台配备四块 NVIDIA H100 PCIe GPU（H100）的服务器上进行。每块 GPU 配备 80 GB HBM2e 内存，带宽为 2093 GB/s，运行 CUDA 12.1。

4) GRW 工作负载：表 II 展示了实验中评估的图数据集，并按边数升序排列。这些数据集涵盖了多种真实世界图，包括 Web、网络、引用和社交网络。表格的第五列（δ）显示了图的直径，即任意两个顶点之间最长的最短路径。

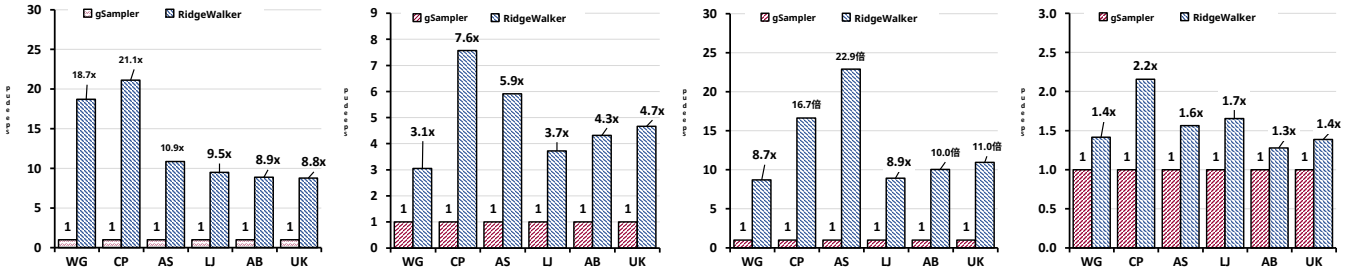
GRW 算法。我们评估了图数据库和图机器学习应用中四种常用的 GRW 算法：PPR [50], URW [49], DeepWalk [5], 和 Node2Vec [9], , 其中 DeepWalk 和 Node2Vec 广泛用于图神经网络工作负载。为与现有基准测试保持一致 [15]–[17], , 我们将查询长度设为 80。对于 Node2Vec, 参数设为 $p = 2$ 和 $q = 0.5$，与其他最先进工作中的设置相同 [15]–[17]。边权重根据 Thunderrw 方法生成 [16]。为与现有系统进行公平比较，我们始终采用相同的最先进采样算法和配置，不改变其算法语义。

性能指标。我们根据有效吞吐量评估 GRW 的执行性能，该指标不受输入查询数量和图规模等工作负载特征变化的影响。吞吐量以每秒百万步（百万步/秒）为单位，通过将访问的顶点总数除以端到端查询处理时间计算得出。处理时间的测量方式如下：首先预热每个 GRW 系统，并将其作为运行时服务运行，同时将输入图预加载到主存中。查询以连续数据流的形式发出，以模拟真实世界应用的行为。每项实验使用相同的查询输入重复五次，并报告中位数结果，以减少异常值的影响。

B. 与最先进的基于 *FPGA* 的解决方案比较

图 8a 比较了 Ridge-Walker 与 FastRW [22], （一种最先进的 GRW 加速器）的 DeepWalk 吞吐量。由于 FastRW 的代码未公开，我们在相同的硬件平台 Alveo U50 现场可编程门阵列上实现了 RidgeWalker，并使用其论文中报告的数据集比较性能。在所有数据集上，RidgeWalker 的表现均优于 FastRW，且随着图规模增大，加速比也随之提高。在最大的数据集（*LJ*）上，RidgeWalker 实现了 $70.98\times$ 的提升。这是因为 FastRW 依赖缓存，将小图存储在容量有限的片上存储器中；但对于大规模图，由于 GRW 本身局部性较差，这种方法效果不佳。相比之下，RidgeWalker 针对高带宽内存中大规模图的随机访问进行了优化。

即使对于 *WG* 这样的小型图，FastRW 可以将图的大部分缓存到片上高速内存中，RidgeWalker 仍能实现 $2.24\times$ 的加速比。这归功于我们的乱序执行，以及使用 ThundeRiNG [48], 进行的流水线化伪随机数生成，后者避免了额外的 HBM 流量。相比之下，FastRW 会预先生成随机数



（a）个性化PageRank　（b）均匀随机游走　（c）DeepWalk　(d) Node2Vec

图9：四种图随机游走应用中相对于 gSampler 的归一化吞吐量对比。

并将其存储在中央处理器上，然后必须将其加载到高带宽内存中，从而占用原本可用于访问图的带宽。

图8b对比了在*WG* 图上运行 PPR 和 URW 时，本研究提出的加速器与 Su 等人 [23]方案的吞吐量。由于他们的代码和评估仅限于这一数据集，我们相应地限制了比较范围。RidgeWalker 在 PPR 和 URW 上的吞吐量分别高出$9.21\times$和 $9.94\times$ ，这主要得益于其高效的内存子系统，能够充分利用高带宽内存带宽。

图8c对比了 RidgeWalker 与使用蓄水池采样的 LightRW [18], 在 Node2Vec 上的吞吐量；蓄水池采样是 LightRW 使用的相同采样方法，二者均在 Alveo U250 FPGA 上实现。RidgeWalker 的性能高出 $1.1\times$–$1.5\times$。虽然两种设计都很高效，但 LightRW 使用批量执行，这会导致流水线停顿。相比之下，RidgeWalker 采用细粒度调度，消除了气泡，并实现了更高的资源利用率。

在加权图上的元路径随机游走中也观察到了类似趋势。如图 8d 所示，得益于更高的提前终止概率，RidgeWalker 相较于 LightRW 实现了 $1.3\times$ 至 $1.7\times$ 的加速比。在元路径游走中，下一跳必须匹配特定类型 [16]；如果不存在这样的邻居，游走就会提前结束。RidgeWalker 利用零气泡调度器 在这种运行时不规则性下保持较高的流水线利用率，始终优于 LightRW。

C. 与当前最先进的基于 *GPU* 的解决方案比较

1) 真实世界图上的分析：图 9a 显示，在全部六个图上，RidgeWalker 在 PPR 上均以 $8.8\times$–$21.1\times$ 的加速比超越 gSampler，并在 CP 上取得了 $21.1\times$ 的最高增益。关键差异在于 RidgeWalker 的零气泡调度器。通过每个周期重新路由就绪任务，它即使在 PPR 游走以差异极大的长度终止时，也能让所有流水线保持繁忙，从而维持完全填充的流水线和接近峰值的随机访问内存带宽。虽然 gSampler 依靠超级批处理 来降低图形处理器上的内核启动开销，但每个线程束仍必须等待最慢的线程。当随机游走提前结束时，空闲线程会产生显著的分歧开销。

图 9b 报告了 RidgeWalker 在 URW 上相对于 gSampler 的加速比。在所有数据集上，RidgeWalker 都更快，加速幅度从 $3.1\times$ 在 WG 上比 $7.6\times$在 CP 上。最大

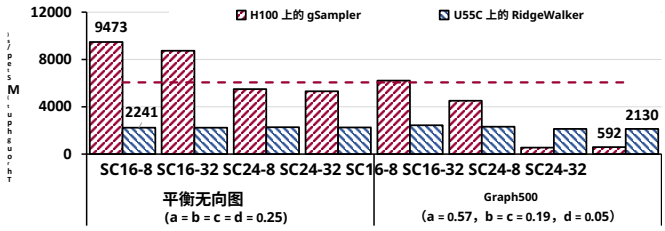


图 10：在均衡配置和 Graph500配置下，RMAT 图上的性能比较。

CP 和 AS 上的性能提升主要得益于我们的异步内存访问引擎充分利用了随机访问带宽。WG 上的提升较小，反映出其规模紧凑，大部分内容都能放入 GPU 缓存；而 LJ 上的提升适中，则是因为其无向结构减少了工作负载不均衡。即便如此，RidgeWalker 在各类图上都实现了稳定的加速比。

图 9c 比较了 RidgeWalker 与 gSampler 在 DeepWalk 上的表现。RidgeWalker 的加速比从 $8.7\times$ (WG) 到 $22.9\times$(AS) 不等；在 CP 和 AB 上，增益超过 $10\times$ 。DeepWalk 依赖别名采样，这使伪随机数的数量翻倍，并增加了图形处理器指令数，将 gSampler 的带宽利用率限制在峰值带宽的 0.9–2.4%。RidgeWalker 将采样和随机数生成完全流水线化，实现了 URW 级吞吐量。这些结果凸显了流水线优化和带宽利用对于大规模图嵌入的重要性。

图 9d 展示了 RidgeWalker 在使用拒绝采样的 Node2Vec 上相较于 gSampler 的加速比 [15]。此处加速比较为温和，从 AB 上的 $1.28\times$ 到 CP 上的 $2.16\times$ 不等。这一结果符合预期，因为 Node2Vec 的偏置游走会在邻居列表上引入更有规律的顺序访问，使图形处理器硬件能够通过批量访问利用局部性。

2）合成图分析：为探究并进一步剖析 RidgeWalker 的性能优势，我们在 H100 GPU 上将 gSampler 中的 DeepWalk 与 RidgeWalker 进行比较，使用具有不同规模和密度的合成 RMAT [55] 图。评估了两种初始配置：平衡的无向配置（RMAT 概率分布设为 a=b=c=d=0.25）和 Graph500 配置（a=0.57, b=c=0.19, d=0.05） [56]。每个图均标记为 SCX-Y，其中 X 表示尺度因子， Y 表示边

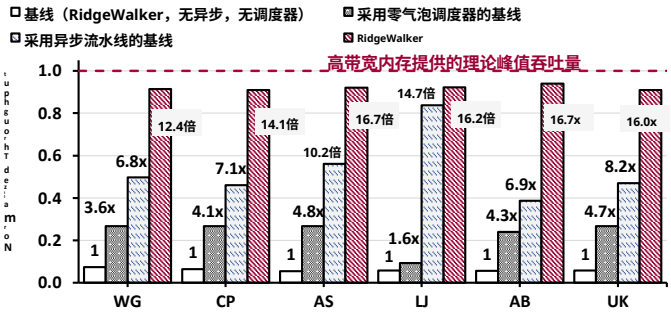


图11：异步流水线和零气泡调度器带来的性能提升分解

因子（例如，SC24-32 对应尺度因子 24 和边因子 32）。

图10中的红色虚线表示基于基准测试得出的 H100 GPU 上 GRW 吞吐量上限，该上限根据实测随机访问内存带宽（不计缓存效应）计算得出。该上限源自随机访问带宽基准测试 [57]。在 SC24平衡图上，gSampler 的吞吐量接近这条线，表明当访问均匀分布时，GPU执行能够达到接近峰值的随机访问效率。

尽管图形处理器在平衡的 RMAT 图上实现了较高的绝对吞吐量，但在偏斜的 Graph500配置下，情况会发生巨大变化。Graph500 会造成严重的结构不平衡，使不同查询的遍历长度差异显著。在 SIMT执行模型下，图形处理器线程束以锁步方式执行，因此提前完成的线程必须等待最长的游走完成。这会导致严重的分歧和资源利用不足，与平衡情况相比，吞吐量降低一个数量级以上。增加查询数量也无济于事，因为已经发起了数百万次并发游走，且所有流式多处理器都已完全饱和。

RidgeWalker 在所有 RMAT 配置下都能保持稳定的高吞吐量。我们的无状态任务分解和零气泡调度实现了完全独立的逐跳执行，使短游走和长游走能够并行进行，互不阻塞。因此，即使面对倾斜严重的图，Ridge Walker 仍能实现约 2,130 MSteps/s 的吞吐量，这表明，对工作负载不均衡的架构容忍能力可能胜过原始带宽优势。这凸显了我们设计的一项关键优势：RidgeWalker 将不规则 GRW 工作负载转化为完全流水线化的执行，从而充分发挥硬件潜力。

D. 分解评估

图11展示了所有真实世界图的归一化吞吐量，所有数值均相对于理论上的 HBM 支持吞吐量（由红色虚线表示，并根据公式（1）计算）。每个柱状条均体现了实际达到的性能，以及相对于我们的分解基线的加速比。该基线保留了 RidgeWalker 架构的整体设计，但禁用了异步执行和动态调度：查询被静态分配到固定的

表III：不同内存配置下，所有图数据集在现场可编程门阵列上的平均 URW 吞吐量

	U250	VCK5000	U50	U55C
内存类型	DDR4	DDR4-NoC	HBM2	HBM2
带宽（GB/s）	77	102	316	460
内存通道数	4	4	32	32
吞吐量（百万步/秒）	258	202	1463	2098
带宽利用率	81%	87%	88%	88%

流水线中，内存请求通过标准 AXI 接口直接发送至高带宽内存，执行则以批量同步的批次进行，不处理提前终止，类似于 LightRW [18] 和 FastRW [22]。随后，我们逐步启用各项优化，以单独评估其贡献。

首先，仅启用零气泡调度器即可将提前终止的查询动态重新分配到各条流水线中。这消除了流水线气泡，使性能提升 1.6×–4.8×。对于*LJ*而言，由于其无向结构导致提前终止的情况极少，性能提升幅度较小。鉴于 KONECT 中全部 418 个图的近 80% [58]为有向图，零气泡调度器对于在实际 GRW 工作负载中维持高利用率至关重要，直接解决了观察 #2，并减轻了提前终止对 GRW 查询造成的显著性能损失。

其次，在禁用调度器的同时启用异步流水线和异步内存访问引擎，可以隔离延迟隐藏与乱序任务执行的影响。该设计使基线性能提升 6.8× 至 14.7×，表明解耦内存访问和逐跳任务独立性对于缓解 GRW 执行中的内存延迟瓶颈至关重要。相较于先前工作，它通过在多个并发查询之间高效摊销指针追逐延迟，解决了观察 #1。

最后，同时启用这两项优化，可在海量并发查询之间实现完美的流水线执行。在此配置下，RidgeWalker 相较基线可实现最高 16.7× 加速比，并达到理论 HBM 随机访问峰值的最高 88%。

E. 对不同 *FPGA* 的支持

为展示 RidgeWalker 架构的通用性，我们在四种 FPGA 平台上评估其性能：U250、VCK5000、U50 和 U55C。这些设备的可用内存带宽和独立内存通道数量各不相同，如表III的第二行和第三行所示。我们测量了 URW 在表 II 所列所有图数据集上的吞吐量（第四行）和随机访问带宽利用率（第五行），并报告其平均值。Versal VCK5000 配备了硬化的片上网络（NoC）内存子系统，包含四个 DDR4 通道，可提供高达 102 GB/s 的总带宽。为支持不规则访问模式，我们关闭了默认的 NoC 通道交错（该功能通常针对顺序负载进行优化），并在 RidgeWalker 中部署了四条处理流水线。总体而言，RidgeWalker 在所有评估的 FPGA 平台上均实现了 81% 至 88% 的带宽利用率。RidgeWalker

表 IV：U55C FPGA 上不同 GRW 的硬件资源消耗（百分比）和频率（MHz）。

App.	LUTs	REGs	块 RAM	DSPs	频率
PPR	61。1%	29。8%	19。5%	2.2%	320兆赫兹
URW	50。1%	24.0%	19。5%	2.2%	320兆赫兹
DeepWalk	67.5%	32.3%	39。1%	4.4%	320兆赫兹
Node2Vec	79。1%	41。6%	36.0%	7.3%	320兆赫兹

仅需支持标准 AXI 协议的独立内存通道，便于在不同 FPGA 之间移植。

F. 资源利用率与频率优化

表 IV 报告了在 U55C FPGA 上实现的四种 GRW 内核中 RidgeWalker 的资源利用率和工作频率。由于 GRW 本质上受内存限制，设计重点在于充分利用可用的随机访问内存带宽和通道，而非耗尽逻辑资源，从而为后续加速器留出充足容量。资源使用情况会随采样方法以及每个行指针条目的大小而变化（RP_{entry}，见第 VII 节）。得益于异步执行模型，各运算符和模块彼此解耦，简化了时序收敛，并支持高达 320 MHz 的频率。

RidgeWalker 使用查找表实现浅 FIFO，从而简化高频运行下的布局布线。较深的缓冲区则使用块RAM，因为每个块最多可存储 512 个条目。两个组件依赖基于块RAM的 FIFO：(1) 异步内存访问引擎中的一个 128 条目元数据队列，其容量足以吸收 HBM 往返延迟；(2) 调度器与流水线之间的 65 条目 FIFO，用于维持稳定的查询处理并防止停顿。

我们进一步优化了零气泡调度器，防止蝴蝶互连网络成为关键路径。我们插入寄存器以打断较长的组合逻辑路径，并将整个模块设计为自由运行模块，无需使用启动/停止触发器等全局控制信号。这消除了高扇出广播逻辑，有助于提升时序表现，从而支持更高的频率。对 U55C FPGA 进行的独立性能分析表明，该调度器的运行频率最高可达 450 MHz，而仅消耗 1.8% 的可用查找表，展现出其扩展至 32 个以上 HBM通道的潜力。

IX. 相关工作与讨论

ASIC 与存内设计。 多项研究通过将计算移近数据，来应对随机游走受内存限制的特性。FlashWalker [59]将游走逻辑嵌入固态硬盘，利用 NAND 层级的并行性消除主机与设备之间的数据传输。其他研究则探索利用 ReRAM 等新兴技术实现存内处理 (PIM)，使图处理能够直接在存储阵列中进行，从而减少数据搬移和延迟 [60]。然而，目前的 PIM 设计仅支持很小的图规模，可扩展性仍然有限。尽管 ASIC 和 PIM 方案具有提升性能的潜力，但它们难以适应不断演进的算法，且开发周期较长基于现场可编程门阵列的解决方案部署灵活，能够适应 GRW 算法的快速演进。

图处理超级计算架构。通用多线程架构，例如 Cray XMT（最初名为 Tera MTA） [61], [62]和 Lucata Emu [63]，[64] ，已被用于探索包括广度优先搜索和单源最短路径在内的不规则图处理工作负载。这些系统通过维持数百个硬件线程上下文，并在每个时钟周期从任一就绪线程发出指令，来容忍内存延迟。RidgeWalker 采取了不同的方法，采用针对特定领域的架构，专为 GRW 设计。利用马尔可夫性质，它使查询任务能够以细粒度、无状态单元的形式直接流经流水线，从而实现乱序执行，无需在内存中维护线程上下文或全局游走状态。此外，RidgeWalker 采用了一个以排队论为形式化依据的调度器，即使面对高度多样且不均衡的 GRW工作负载，也能维持接近最优的吞吐量。

稀疏数据结构取数器。Widx [65] 采用有状态设计加速数据库哈希索引查找，将指针追踪和键哈希卸载到可编程片上单元，并在分发器表和本地缓存中为每个键维护状态。Fifer [66] 采用粗粒度的分时复配置，以容忍内存延迟并平衡不规则工作负载。TMU [67]面向数据移动密集型张量运算符，利用可重构地址抽象进行粗粒度和细粒度的数据重排。Terminus [68] 通过维护每次操作和每个分区的状态，扩展了稀疏数据结构加速能力，以支持哈希表和树上的细粒度更新。Aurochs [69]将数据流加速器扩展为通过有状态细粒度硬件线程执行不规则结构。这些设计主要采用有状态方式，在寄存器、缓存或缓冲区中维护每个查询或任务的上下文，以重叠内存访问与计算。相比之下，RidgeWalker 采用基于马尔可夫性质的无状态任务分解模型，从而实现灵活调度和大规模并行性。该架构专为 GRW 工作负载设计，可实现近乎最优的随机访问吞吐量，并为新兴的大规模图应用提供高效率。

讨论。RidgeWalker 采用标准架构原语，而非 FPGA 专用特性，因此可以轻松移植到专用集成电路，并与现有逻辑相辅相成。此外，我们的完美流水线化策略可推广到其他概率型工作负载，例如贝叶斯网络和马尔可夫链蒙特卡罗应用；在这些应用中，运行时依赖关系和随机访问延迟会显著影响性能。

第十节 结论

本文介绍了 RidgeWalker，这是一种完美流水线化的基于现场可编程门阵列的加速器，它利用 GRW 的马尔可夫性质，将查询分解为无状态任务。RidgeWalker 将多个异步流水线与所提出的现场可编程门阵列零气泡调度器相结合，最大限度地发挥并行性，使运行时利用率达到理论随机访问内存带宽的 88%。RidgeWalker 的性能提升达到

比当前最先进的现场可编程门阵列和图形处理器解决方案分别快 71.0× 和 22.9× 。此外，RidgeWalker 的模块化设计支持多种 GRW 算法，为高性能 GRW 提供了可扩展且高效的架构。

致谢

本研究/项目获得新加坡教育部 AcRF Tier 2（编号：MOE-T2EP20224-0020）和 Tier 1（编号：T1 251RES2315）资助、Google 2025年南亚与东南亚研究奖，以及加拿大自然科学与工程研究委员会（NSERC）探索基金的支持。我们还感谢 AMD异构加速计算集群（HACC）计划慷慨捐赠硬件 [70] 。新加坡国立大学的 Yao Chen 为通讯作者。

参考文献

[1] R. Lambiotte、J.-C. Delvenne 和 M. Barahona，“随机游走、马尔可夫过程与复杂网络的多尺度模块化组织”，*IEEE*网络科学与工程汇刊，第 1 卷，第 2 期，第 76–90 页，2015 年。[2] Y. Shao、S. Huang、X. Miao、B. Cui 和 L. Chen，“面向大规模图高效二阶随机游走的内存感知框架”，载于*2020* 年 *ACM SIGMOD* 数据管理国际会议论文集，SIGMOD ’20 系列。美国纽约：计算机协会，2020 年，第 1797–1812 页。 [在线]。获取地址：https://doi.org/ 10 1145 3318464 3380562 。/。[3] M. Liao、R.-H. Li、Q. Dai、H. Chen、H. Qin 和 G. Wang，“高效个性化 PageRank 计算：方差缩减蒙特卡罗方法的优势”，*ACM* 数据管理会议论文集，第 1 卷，第 2 期，第 1–26 页，2023 年。[4] X. Yu、X. Ren、Y. Sun、Q. Gu、B. Sturt、U. Khandelwal、B. Norick 和 J. Han，“个性化实体推荐：一种异构信息网络方法”，载于第七届 *ACM Web* 搜索与数据挖掘国际会议论文集，2014 年，第 283–292 页。[5] B. Perozzi、R. Al-Rfou 和 S. Skiena，“Deepwalk：社会表示的在线学习”，载于第 *20* 届 *ACM SIGKDD* 知识发现与数据挖掘国际会议论文集，2014 年，第 701–710 页。[6] J. J. Miller，“基于 Neo4j 的图数据库应用与概念”，载于美国佐治亚州亚特兰大举行的南方信息系统协会会议论文集，第 2324 卷，2013 年，第 141–147 页。[7] Z. Li、D. Fu 和 J. He，“个性化 PageRank 中万物皆在演化”，载于*2023* 年 *ACM Web* 会议论文集，WWW ’23 系列。美国纽约：计算机协会，2023 年，第 3342–3352 页。 [在线]。获取地址：https://doi.org/10.1145/3543507。3583474[8] Y. Dong、N. V. Chawla 和 A. Swami，“MetaPath2Vec：面向异构网络的可扩展表示学习”，载于第 *23* 届 *ACM SIGKDD* 知识发现与数据挖掘国际会议论文集，2017 年，第 135–144 页。[9] A. Grover 和 J. Leskovec，“Node2Vec：面向网络的可扩展特征学习”，载于第 *22* 届 *ACM SIGKDD* 知识发现与数据挖掘国际会议论文集，2016 年，第 855–864 页。[10] L. Cappelletti、T. Fontana、E. Casiraghi、V. Ravanmehr、T. J. Callahan、C. Cano、M. P. Joachimiak、C. J. Mungall、P. N. Robinson、J. Reese 等，“Grape：用于快速、可扩展图处理和基于随机游走的嵌入”，《自然·计算科学》，第 3 卷，第 568 页，2023 年，第 6 期，第 552 页。[11] M. Wang 、 L. Yu 、 D. Zheng 、 Q. Gan 、 Y. Gai 、 Z. Ye 、 M. Li、J. Zhou、Q. Huang、C. Ma、Z. Huang、Q. Guo、H. Zhang、H. Lin、J. Zhao、J. Li、A. J. Smola 和 Z. Zhang，“深度图学习库：迈向高效、可扩展的图上的深度学习”，*CoRR*，第 abs/1909.01315 卷，2019 年。[在线]。获取地址：http://arxiv.org/abs/1909.01315

[12] Q. Liu、L. Jiang、M. Han、Y. Liu 和 Z. Qin，“知识图谱中的层次化随机游走推理”，载于第39届国际 ACM SIGIR 信息检索研究与开发会议论文集，2016年，第445–454页。[13] X. Wang、X. He 和 T.-S. Chua，“面向推荐的图学习与推理”，载于第13届国际网络搜索与数据挖掘会议论文集，2020年，第890–893页。[14] B. Jimenez Gutierrez、Y. Shu、Y. Gu、M. Yasunaga 和 Y. Su，“Hipporag：受神经生物学启发的大语言模型长期记忆”，神经信息处理系统进展，第37卷，第59 532–59 569页，2024年。[15] P. Gong、R. Liu、Z. Mao、Z. Cai、X. Yan、C. Li、M. Wang 和 Z. Li，“gSampler：用于图学习的通用高效基于 GPU 的图采样”，载于第29届操作系统原理研讨会论文集，2023年，第562–578页。[16] S. Sun、Y. Chen、S. Lu、B. He 和 Y. Li，“Thunderrw：内存图随机游走引擎”，VLDB基金会会议论文集，第14卷，第11期，第1992–2005页，2021年。[17] K. Yang、M. Zhang、K. Chen、X. Ma、Y. Bai 和 Y. Jiang，“Knightking：快速分布式图随机游走引擎”，载于第27届 ACM 操作系统原理研讨会论文集，2019年，第524–537页。[18] H. Tan、X. Chen、Y. Chen、B. He 和 W.-F. Wong，“LightRW：FPGA加速的动态图随机游走”，ACM 数据管理论文集，第1卷，第1期，2023年5月。 [在线]。获取地址：https://doi.org/10.1145/ 3588944[19] H. Yin、Y. Shao、X. Miao、Y. Li 和 B. Cui，“利用压缩图在 GPU 上进行可扩展图采样”，载于第31届 ACM 信息与知识管理国际会议论文集，2022年，第2383–2392页。[20] A. Jangda、S. Polisetty、A. Guha 和 M. Serafini，“利用 GPU 加速面向图机器学习的图采样”，载于第16届欧洲计算机系统会议论文集，2021年，第311–326页。[21] P. Wang、C. Xu、C. Li、J. Wang、T. Wang、L. Zhang、X. Hou 和 M. Guo，“优化基于 GPU 的图采样与随机游走以提高效率和可扩展性”，IEEE 计算机汇刊，第72卷，第9期，第2508–2521页，2023年。[22] Y. Gao、T. Wang、L. Gong、C. Wang、X. Li 和 X. Zhou，“FastRW：面向 FPGA 图随机游走的数据流高效且具备内存感知能力的加速器”，载于2023年欧洲设计、自动化与测试会议暨展览会（DATE）。IEEE，2023年，第1–6页。[23] C. Su、H. Liang、W. Zhang、K. Zhao、B. Ai、W. Shen 和 Z. Wang，“利用快速随机游走器在配备高带宽内存的 FPGA 加速器上进行图采样”，载于2021年第31届国际现场可编程逻辑与应用会议（FPL）。IEEE，2021年，第211–218页。[24] X. Chen、H. Tan、Y. Chen、B. He、W.-F. Wong 和 D. Chen，“ThunderGP：基于 HLS 的 FPGA 图处理框架”，载于2021年 ACM/SIGDA 国际现场可编程门阵列研讨会，2021年，第69–80页。[25] X. Chen、Y. Chen、F. Cheng、H. Tan、B. He 和 W.-F. Wong，“Regraph：利用异构流水线在配备高带宽内存的 FPGA 上扩展图处理”，载于2022年第55届 IEEE/ACM 国际微体系结构研讨会（MICRO）。IEEE，2022年，第1342–1358页。[26] L. Stasytis 和 Z. Istv´an，“FPGA 上 Hestenes-Jacobi SVD 的优化技术”，载于2023年第33届国际现场可编程逻辑与应用会议（FPL）。IEEE，2023年，第144–150页。[27] S. Dave、Y. Kim、S. Avancha、K. Lee 和 A. Shrivastava，“DMazerunner：在数据流加速器上执行完全嵌套循环”，ACM 嵌入式计算系统汇刊（TECS），第18卷，第5s期，第1–27页，2019年。[28] J. Davis 和 R. Reese，《有限状态机数据通路设计、优化与实现》。Springer Nature，2022年。[29] Z. Zhang 和 B. Liu，“基于 SDC 的模调度用于流水线综合”，载于2013年 IEEE/ACM 国际计算机辅助设计会议（ICCAD），2013年，第211–218页。[30] W. Jaiyeoba、N. Elyasi、C. Choi 和 K. Skadron，“ACTS：近内存 FPGA 图处理框架”，载于2023年 ACM/SIGDA 国际现场可编程门阵列研讨会论文集，2023年，第79–89页。[31] O. Jaiyeoba 和 K. Skadron，“Dynamic-acts：面向配备高带宽内存的 FPGA 的动态图分析加速器”，ACM 可重构技术与系统汇刊，第17卷，第3期，第1–29页，2024年。

[32] O. Jaiyeoba、A. T. Mughrabi、M. Baradaran、B. Gul 和 K. Skadron，“Swift：用于扩展加速图分析的多FPGA框架”，2024年。[33] S. Zhou、R. Kannan、V. K. Prasanna、G. Seetharaman 和 Q. Wu，“HitGraph：FPGA上的高吞吐图处理框架”，IEEE并行与分布式系统汇刊，第30卷，第10期，第2249–2264页，2019年。[34] Y. Hu、Y. Du、E. Ustun 和 Z. Zhang，“GraphLily：在配备高带宽内存的 FPGA 上加速图线性代数”，载于2021年 IEEE/ACM 国际计算机辅助设计会议（ICCAD）。IEEE，2021年，第1–9页。[35] S. Rahman、N. Abu-Ghazaleh 和 R. Gupta，“GraphPulse：用于异步图处理的事件驱动硬件加速器”，载于第53届 IEEE/ACM 国际微体系结构研讨会（MICRO-53）论文集，2020年，第908–921页。[36] S. Rahman、M. Afarin、N. Abu-Ghazaleh 和 R. Gupta，“JetStream：利用事件驱动硬件加速器对流式数据进行图分析”，载于第54届 IEEE/ACM 国际微体系结构研讨会（MICRO-54）论文集，2021年，第1091–1105页。[37] P. Yao 等，“Scalagraph：用于大规模并行图处理的可扩展加速器”，载于第28届 IEEE 高性能计算机体系结构国际研讨会（HPCA）论文集，2022年，第199–212页。[38] X. Chen、F. Cheng、H. Tan、Y. Chen、B. He、W.-F. Wong 和 D. Chen，“ThunderGP：基于 HLS 的 FPGA 资源高效图处理框架”，ACM 可重构技术与系统汇刊（TRETS），2022年。[39] K. Asifuzzaman、M. Abuelala、M. Hassan 和 F. J. Cazorla，“揭秘高带宽内存在实时系统中的特性”，载于2021年 IEEE/ACM 国际计算机辅助设计会议（ICCAD）。IEEE，2021年，第1–9页。[40] V. Kanade、F. Mallmann-Trenn 和 T. Sauerwald，“图中的合并时间：合并何时能像相遇一样快？”，ACM 算法汇刊，第19卷，第2期，第1–46页，2023年。[41] R. I. Oliveira 和 Y. Peres，“图上的随机游走：关于命中、相遇、合并和返回的新界”，载于2019年分析算法学与组合数学研讨会（ANALCO）论文集。SIAM，2019年，第119–126页。[42] O. Moreira、A. Yousefzadeh、F. Chersi、A. Kapoor、R.-J. Zwartenkot、P. Qiao、G. Cinserin、M. A. Khoei、M. Lindwer 和 J. Tapson，“Neuronflow：用于人工智能工作负载的混合神经形态-数据流处理器架构”，载于2020年第2届 IEEE 人工智能电路与系统国际会议（AICAS）。IEEE，2020年，第01–05页。[43] J. H. Dshalalow，“经典排队方法选编”，载于《排队论、方法与开放问题进展》。CRC Press，2023年，第1–42页。[44] Y. Lu、T. Abdelzaher、C. Lu、L. Sha 和 X. Liu，“利用排队论预测进行反馈控制，以保证 Web服务器的相对延迟”，载于第9届 IEEE 实时与嵌入式技术及应用研讨会，2003年。会议论文集，2003年，第208–217页。[45] Y. Sun、C. E. Koksal 和 N. B. Shroff，“多服务器系统中批作业的近似延迟最优调度”，2023年。 [在线]。获取地址：https://arxiv.org/abs/ 2309 16880 。F. D. Croce 和 R. Scatamacchia，“同构并行机器的最长处理时间规则再探”，2018年。 [在线]。获取地址：https://arxiv.org/abs/ 1801 05489 。P. Yu、Y. Qiu、X. Jin 和 M. Chowdhury，“Orloj：可预测地服务不可预测的深度神经网络”，2022年。 [在线]。获取地址：https://arxiv.org/abs/ 2209.00159[48] H. Tan、X. Chen、Y. Chen、B. He 和 W.-F. Wong，《ThundeRiNG：在 FPGA 上生成多个独立随机数序列》。美国纽约州纽约：计算机协会，2021年，第115–126页。 [在线]。获取地址：https://doi.org/10.1145/3447818。3461664[49] R.-H. Li、J. X. Yu、L. Qin、R. Mao 和 T. Jin，“基于随机游走的图采样”，载于2015年 IEEE 第31届国际数据工程会议。IEEE，2015年，第927–938页。[50] G. Hou、X. Chen、S. Wang 和 Z. Wei，“个性化PageRank的大规模并行算法”，VLDB基金会会议论文集，第14卷，第9期，第1668–1680页，2021年。[51] A. J. Walker，“生成具有任意频率分布的离散随机数的新型快速方法”，《电子学快报》，第8卷，第10期，第127–128页，1974年。[52] F. Zhang、M. Jiang 和 S. Wang，《高效动态加权集合采样及其扩展》，VLDB基金会会议论文集，第17卷，第1期，第15–27页，2023年。[53] J. Leskovec 和 A. Krevl，《SNAP数据集：斯坦福大型网络数据集集合》，http://snap.stanford.edu/data，2014年6月。[54] P. Boldi 和 S. Vigna，《WebGraph框架 I：压缩技术》，载于第十三届国际万维网会议（WWW 2004）论文集。美国曼哈顿：ACM出版社，2004年，第595–601页。[55] D. Chakrabarti、Y. Zhan 和 C. Faloutsos，《R-mat：用于图挖掘的递归模型》，载于2004年SIAM国际数据挖掘会议论文集。SIAM，2004年，第442–446页。[56] R. C. Murphy、K. B. Wheeler、B. W. Barrett 和 J. A. Ang，《介绍Graph 500，22，2010 19，45 74》，Cray用户组（CUG），第 卷，第 期，第 页。[57] C. Lutz、S. Breß、S. Zeuch、T. Rabl 和 V. Markl，《提升音量：借助高速互连在GPU上处理大规模数据》，载于SIGMOD。美国纽约州纽约：ACM，2020年，第1633–1649页。[58] J. Kunegis，《KONECT：科布伦茨网络集合》，载于第22届国际万维网会议论文集，2013年，第1343–1350页。[59] F. Niu、J. Yue、J. Shen、X. Liao、H. Liu 和 H. Jin，《FlashWalker：用于图随机游走的存储内加速器》，载于2022年IEEE国际并行和分布式处理研讨会（IPDPS）。IEEE，2022年，第1063–1073页。[60] D. Choudhury、L. Xiang、A. Rajam、A. Kalyanaraman 和 P. P. Pande，《在支持3D NoC的近存处理架构上加速图计算》，ACM电子系统设计自动化汇刊，第28卷，第3期，第1–16页，2023年。[61] D. A. Bader 和 K. Madduri，《为Cray MTA-2设计广度优先搜索和ST连通性的多线程算法》，载于2006年国际并行处理会议（ICPP’06）。IEEE，2006年，第523–530页。[62] D. Mizell 和 K. Maschhoff，《大型Cray XMT系统的早期经验》，载于2009年IEEE国际并行与分布式处理研讨会。IEEE，2009年，第1–9页。[63] E. R. Hein、S. Eswar、A. Yas¸ar、J. Li、J. S. Young、T. M. Conte、¨U. V. C¸ataly¨urek、R. Vuduc、J. Riedy 和 B. Uc¸ar，《Emu Chick上不规则算法的编程策略》，ACM并行计算汇刊（TOPC），第7卷，第4期，第1–25页，2020年。[64] E. Smith、S. Kuntz、J. Riedy 和 M. Deneroff，《Lucata Pathfinder上的并发图查询》，2022年。 [在线。可用地址：https://arxiv.org/abs/ 2209 11889 。O. Kocberber、B. Grot、[65] J. Picorel、B. Falsafi、K. Lim 和 P. Ran-ganathan，《认识游走体：加速内存数据库中的索引遍历》，载于第46届IEEE/ACM国际微体系结构研讨会论文集，2013年，第468–479页。[66] Q. M. Nguyen 和 D. Sanchez，《Fifer：在可重构架构上实用地加速不规则应用》，载于MICRO-54：第54届IEEE/ACM国际微体系结构研讨会，2021年，第1064–1077页。[67] W. Zhou、Z. Wang、C. Chen、Y. Li、Y. Yang、Z. Wu 和 A. Chat-topadhyay，《张量操作单元（TMU）：面向高吞吐量AI片上系统的可重构近存张量操作》，arXiv预印本 arXiv:2506.14364，2025年。[68] H. R. Lee 和 D. Sanchez，《Terminus：用于稀疏数据结构读写操作的可编程加速器》，载于2024年第57届IEEE/ACM国际微体系结构研讨会（MICRO）。IEEE，2024年，第1233–1246页。[69] M. Vilim、A. Rucker 和 K. Olukotun，《Aurochs：一种数据流线程架构》，载于2021年ACM/IEEE第48届国际计算机体系结构研讨会（ISCA）。IEEE，2021年，第402–415页。[70] AMD，《异构加速计算集群（HACC）计划》，https://www.amd-haccs.io/index.html，2023年。