计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(5): 1209−1223, 2026

使用自适应缓存结构优化图计算

贾朝阳　陈　湜　刘京雨　沈　立
（国防科技大学计算机学院　长沙　410073）
（jiachaoyang18@nudt.edu.cn）

Optimizing Graph Computing with an Adaptive Cache Structure

Jia Chaoyang, Chen Shi, Liu Jingyu, and Shen Li
（*College of Computer Science and Technology*, *National University of Defense Technology*, *Changsha* 410073）

Abstract　With the continuous advancement of computer hardware and software, graph computing has been widely applied in many fields, including social network analysis, recommendation systems, bioinformatics, and fraud detection. The memory access behavior in graph computing is highly variable: different kernels accessing property arrays exhibit different spatial locality when graph computing is executed on a GPU, and the same kernel shows varying spatial locality across different property arrays. Existing cache optimization strategies in GPU architectures fail to enhance the performance of graph computing effectively. State-of-the-art cache optimization strategies cannot implement distinct management strategies for data with different reusability, which is an essential reason for the poor performance of graph computing. To address this, AB-Cache architecture is proposed, specifically designed for GPU platforms. AB-Cache adopts an adaptive approach that utilizes two cache structures to optimize memory access requests with different spatial locality characteristics. An online automatic classification mechanism compatible with AB-Cache architecture is also proposed, enabling fast classification of memory access requests with different spatial locality with minimal overhead. Comprehensive evaluations across multiple graph computing applications and a wide range of graph datasets show that AB-Cache scheme achieves a 1.14 times speedup over the baseline, demonstrating its effectiveness and practicality in optimizing graph computing performance.

Key words　graph computing; heterogeneous architecture; general purpose graphics processing unit; cache; locality classification

摘　要　随着计算机硬件与软件的不断进步，图计算的应用领域日益广泛。图计算的访存行为多变，图计算在 GPU 执行时不同 kernel 访问属性数组展现出不同的空间局部性，同一个 kernel 访问不同的属性数组也展现出不同的空间局部性。当前 GPU 架构中的缓存优化策略未能有效挖掘图计算的性能潜力，最先进的缓存优化策略无法针对不同重用性的数据实施不同的管理策略，这是导致图计算性能低下的一个重要原因。为了解决上述问题，提出了专为 GPU 平台设计的 AB-Cache 架构。AB-Cache 采用自适应的思想，巧妙地运用 2 种缓存结构分别优化具有不同空间局部性特征的内存访问请求。与 AB-Cache 架构相契合的在线自动分类机制能够以极低的开销迅速分类具有不同空间局部性的内存访问请求。通过对多个图计算应用及广泛图数据集的综合评估，AB-Cache 相较于基线方案实现了 1.14 倍的加速效果，证明了该方案在图计算性能优化方面的有效性与实用性。

收稿日期：2025-03-18；**修回日期**：2026-01-12
基金项目：国家自然科学基金重点项目（62032001）；国家自然科学基金项目（62472436）
This work was supported by the Key Program of the National Natural Science Foundation of China (62032001) and the National Natural Science Foundation of China (62472436).
通信作者：沈立（lishen@nudt.edu.cn）

关键词 图计算;异构体系结构;通用图形处理器;缓存;局部性分类

中图法分类号 TP333

DOI: 10.7544/issn1000-1239.202550219 **CSTR:** 32373.14.issn1000-1239.202550219

随着社会的发展和技术的进步,图计算的应用得到了广泛拓展,其在路径分析与优化[1]、人工智能算法设计[2-3]、生物医药复杂网络解析[4]、社交网络流量监控与管理[5]等关键领域均展现出深远的影响与不可替代的重要价值。面对数据量的急剧膨胀,例如Facebook的社交网络图每日增量达到TB数量级,图计算任务对计算机系统性能的需求急剧增加。业界倾向于采用分布式图计算系统来加速这类高负载任务。然而,分布式系统导致高昂的通信成本以及复杂的负载均衡挑战。随着半导体技术的飞速进步与图计算应用设计的日益成熟,在单台计算机上高效处理包含数千万甚至数十亿条边的大规模图已经成为可能[6-7]。这种转变标志着图计算不再受分布式架构的限制,可以在单台计算机上执行大规模图计算[8-9]。引入异构计算体系结构可以进一步提高单机的性能。

现实世界的图数据普遍遵循幂律分布[10-11],即少数顶点链接大量的边,而绝大多数顶点仅维持着少量的边。这种特性广泛体现在社交网络、城市道路网络、航空网络等众多领域。高度数顶点由于其庞大的邻居集合,在图计算过程中成为访问的焦点,因此具备较高的重用潜力。相反,低度数顶点因连接稀疏,其重用性相对较低。通过量化分析发现,可以将Twitter图中10%的顶点根据度数归类为高重用顶点,并且与这些顶点相连的边的数量占总边数的83%[12]。幂律图的这种特殊拓扑结构导致图计算在遍历图数据时,访存地址分布离散且难以预测,使得访存行为的空间局部性显著下降。

大型图数据集的规模可达数GB至数十GB,远远超出了现代计算机的缓存(cache)容量,导致严重的缓存未命中(cache miss)与缓存抖动(cache thrashing)。图形处理器(graphics processing unit, GPU)平台的缓存资源相对有限,上述问题显得尤为严峻。幂律图不规则的访存模式进一步增加了缓存系统面临的挑战。本文构建了基于CPU与GPU(英伟达A100)的异构计算平台,并测量了在该平台使用k-Core算法[13-14]处理大型图数据(例如soc-LiveJournal[15])时,图计算的访存行为。本文使用NVIDIA Nsight Compute工具收集实验数据。实验结果表明,GPU的二级缓存利用率极低,单次迭代的缓存命中率仅为4.46%。先前研究表明,DRAM访问时间占据图计算总时间的80%[5]。提升缓存系统的性能已成为优化单机环境下图计算性能的核心策略与关键突破口。

前期研究已广泛探索了多种缓存架构设计与管理策略以提升性能[12,16-17]。诸如RRIP(recency and reuse identification policy)[18]等概率性方法,在捕捉和利用图计算局部性方面表现欠佳。KPC(kill-the-PC)[19]和MPPPB(multiperspective placement, promotion, and bypass)[20]等基于即时状态和瞬时多特征融合的方案均在不同程度上提高了缓存性能。SHiP(signature-based hit predictor)[21], Hawkeye[22-23], Glider[24], RLR(reinforcement learned replacement)[16]等基于历史记录的策略[20-29]能够学习并适应图计算中的数据重用模式,但面对复杂的间接访存模式,其效果往往有限。现有的方案都使用单一缓存结构,难以应对图计算中多种类型的局部性。此外,上述方案多聚焦于CPU优化,而GPU在图计算中遭遇的缓存性能瓶颈更为严峻。

统计数据表明,图计算中最后一级缓存(last level cache, LLC)的访问中78%~94%都是访问属性数组,属性数组导致的LLC未命中占据全部LLC未命中的很大一部分[12]。本文重点关注与属性数组相关的访存请求的空间局部性,单一优化方案往往不能覆盖程序执行的所有情况。在软件优化领域,例如任务调度,为了适应各种不同的需求常常采用自适应的方案,对不同特性的任务使用不同的调度策略。与之相比,计算机体系结构层面缺少类似的优化方案。这是因为,与软件层面相比,在体系结构层面实现自适应优化时,需要为多个优化策略定制不同的硬件,导致成本增加。此外,自适应方案还需要根据执行情况切换优化方案,会导致部分硬件空闲。例如,针对不同的计算任务需要安排不同类型的计算单元,这些计算单元往往难以同时使用。

我们设计并实现了一个名为AB-Cache的混合缓存结构,以优化GPU平台的缓存性能,该结构采用自适应策略,将2种不同结构的Cache(A-Cache与B-Cache)融合在一起,分别优化具有不同空间局部性特征的内存访问请求。A-Cache专为那些较低空间局部性的访存请求而设计,使用细粒度缓存行(cache

line)来减少访问延迟并提供更高的灵活性；B-Cache则采用粗粒度缓存行，更有效地利用图计算的空间局部性。为了精准地分配访存请求至最适合的缓存结构，我们开发了一种在线自动分类算法。该算法基于GPU平台运行时的信息将访存请求划分为两大类：一类是具有相对较低空间局部性、更适合A-Cache处理的请求；另一类则是其余请求，其展现出更好的空间局部性，更适合B-Cache处理。需要注意的是，虽然采用了自适应的思想，AB-Cache并没有浪费缓存容量。与传统缓存一样，GPU执行图计算时，整个AB-Cache都可以被读写。

本文的主要贡献包括3个方面：

1)发现图计算的局部性复杂多变，使用单一缓存结构很难提高性能。GPU执行图计算时不同核函数(kernel)访问属性数组展现出不同的空间局部性，同一个kernel访问不同的属性数组展现出不同的空间局部性。

2)提出了针对GPU平台的AB-Cache结构。AB-Cache采用自适应策略，使用不同缓存结构优化具有不同空间局部性的访存请求。

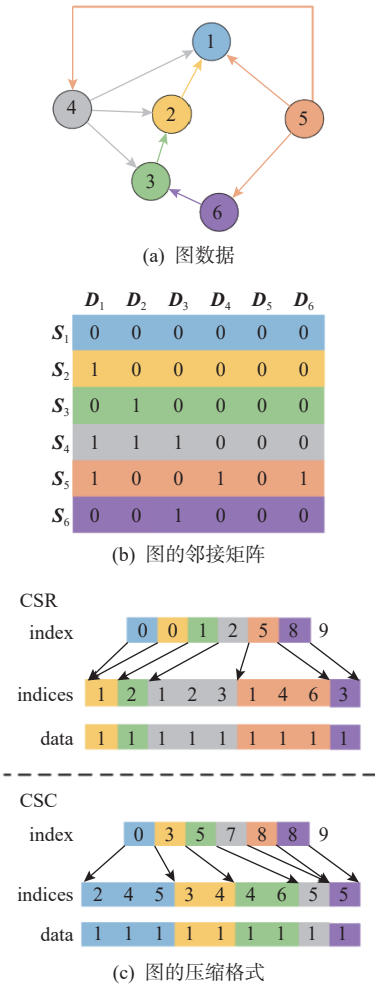
3)提出了针对AB-Cache结构的在线自动分类方案，基于访存请求的空间局部性自动将它们划分为2类。自动分类方案使得AB-Cache执行时无需人工介入。

1 相关背景

1.1 图计算

图以其独特的灵活性，允许任意2个数据元素之间建立直接联系，高效地刻画出顶点间错综复杂的关系。这一优势在诸如社交网络分析等实际应用中展现出巨大的价值[30]。图本质上由顶点、连接这些顶点的边、它们各自携带的属性数据共同构成。为了直观地展现这种结构，邻接矩阵成为一种常用的表示方法，矩阵的行列索引分别映射到图中的源顶点和目标顶点，而矩阵的非0元素表示顶点对之间的连接状态及关系强度。图1(a)和图1(b)为一个包含6个顶点与9条边的简单图的邻接矩阵表示示例。

在现实应用场景中，图普遍呈现出稀疏性[10]，因此图数据通常采用压缩格式保存，其中压缩稀疏行(compressed sparse row, CSR)和压缩稀疏列(compressed sparse column, CSC)格式因其高效性而被广泛使用[31]。图1(c)展示了图1(a)中的简单图在CSR和CSC格式下的表示。CSR格式按行优先顺序进行压缩，通过indices, index, data三个数组实现对邻接矩阵的按行压缩。其中，index数组记录了邻接矩阵中每行的偏移量，indices数组以按行优先的顺序存储每个非0元素的列索引，而data数组则存储矩阵中的非0值。类似地，CSC格式按列优先顺序进行压缩。



(a) 图数据

(b) 图的邻接矩阵

(c) 图的压缩格式

Fig. 1 Representation of a graph

图1 图的表示

1.2 实验发现

在处理海量图数据时，图计算在CPU+GPU构成的单个计算机上执行会遭遇性能瓶颈：大规模数据遍历导致频繁的缓存未命中，缓存中的数据被不断换出，显著拖慢了处理速度。本文聚焦于挖掘图计算过程中的局部性，旨在提升单个计算机缓存的性能表现。通过表1中展示的不同图计算应用在web-Google数据集[15]上的局部性分析，我们探讨了缓存设计的优化方向。我们利用GPGPU-Sim模拟器生成图计算的访存地址流，并将其输入我们开发的缓存模拟器以评估性能。该缓存模拟器仅实现基本的组相联缓存功能，无需完整模拟GPU微架构，即可通过地址流分析访存模式的离散性。GPGPU-Sim的模拟参数与

表 2 中基线缓存的配置参数一致。缓存模拟器除了相联度和容量，其余参数与表 2 中基线缓存的参数一致。

Table 1 Locality Analysis of the Different Graph Computing Applications on web-Google Graph

表 1 不同图计算应用在 web-Google 图上的局部性分析

图计算应用	相联度比例		
	平均值	最小值	最大值
Triangle-Count	9.18	8.66	39.31
k-Core	4.97	4.31	5.69
Graph-Coloring	10.72	10.69	10.75
Degree-Center	8.24	8.22	8.25

Table 2 Experimental Parameters

表 2 实验参数

GPU 组件	参数
GPU 核	每个流多处理器（streaming multiprocessor，SM）包含 64 个浮点单元和 32 个 CUDA 核，总共 8 192 个浮点单元，运行频率为 1.41 GHz。
一级缓存	容量为 8 MB，512 个 set，128 路组相联，缓存行粒度为 128 B，LRU 替换策略，访问延迟 2 个周期。
AB-Cache	A-Cache：容量 20 MB，40 960 个 set，16 路组相联，缓存行粒度 32 B，LRU 替换策略； B-Cache：容量 20 MB，10 240 个 set，16 路组相联，缓存行粒度 128 B，LRU 替换策略； AB-Cache 访问延迟为 15 个周期。
基线缓存	容量 40 MB，20 480 个 set，16 路组相联，缓存行粒度 128 B，LRU 替换策略，访问延迟为 15 个周期。
RRIP	容量 40 MB，20 480 个 set，16 路组相联，缓存行粒度 128 B，RRIP 替换策略，访问延迟为 15 个周期。
DRAM	容量 40 GB，12 个分区，每个分区 2 个子分区，运行频率为 3 500 MHz； 访存粒度为 128 B（B-Cache）时，访存延迟为 450 个周期； 访存粒度为 32 B（A-Cache）时，访存延迟为 300 个周期。

假定 n-路组相联缓存架构，每个缓存行设定为 128 B，总共包含 2 048 个组(set)。这里的 n 并非固定值，而是依据算法与图数据的具体需求动态变化。当缓存的 set 数目固定，缓存容量只与相联度有关。如果确保图计算不发生缓存替换，则针对不同的图计算应用和图数据，每个 set 容纳的缓存行数量 n 也不相同。n 的大小与地址访问模式的离散程度相关：n 愈大，意味着地址访问越加分散。以 k-Core 算法为例，n 在最小情况下可低至 138，但在极端情况下则急剧攀升至 182。平均来看，为了有效利用 web-Google 数据集的局部性，需要将缓存设置为 159 路组相联。这一结果表明，若采用常见的 16 路或 32 路组相联缓存配置，图计算将面临严重的缓存冲突。为了更直观地展示图计算访存行为的离散性，本文将实验数据处理为相对值。本文以常见的 32 路组相联缓存为基准，使用相联度比例(n/32)衡量图计算地址流的离散程度。相联度比例越大，表明图计算的地址流越离散。

随着工艺技术的演进，存储架构的带宽得到了显著提升。为了更有效地利用这一优势，当代处理器倾向于采用更大的缓存行粒度来增强内存带宽的利用效率。分析表 1 中的数据可以直观推断出图计算的空间局部性较差。由于采用较大的缓存行粒度，单个缓存行所承载的数据中，仅有一部分被实际利用，这导致了内存带宽的未充分利用以及缓存容量的不必要消耗。鉴于此，设计更为高效、更具针对性的缓存架构显得尤为重要。

2 AB-Cache

当前优化图计算内存访问性能的方法普遍使用单一的缓存结构。然而，本研究揭示了一个重要现象：在图计算执行过程中，同一个 kernel 对不同的属性数组具有不同的访问行为。进一步地，不同 kernel 之间对属性数组的访问行为差异更大。基于这些见解，本文提出了面向 GPU 平台的 AB-Cache 架构。该架构采用自适应策略，通过部署多样化的缓存结构来分别优化具有不同空间局部性的内存访问请求。AB-Cache 被设计为 GPU 架构的最后一级缓存。

本文采用平均离散距离作为量化指标，评估 kernel 在访问各个属性数组时的空间局部性。这通过追踪并记录访问每个属性数组的地址序列来实现。连续访问某个属性数组时，访存地址序列中 2 个相邻的地址之间的差值被定义为离散距离。在 1 轮迭代内，关于这个属性数组的所有离散距离的平均值称之为这个属性数组在该轮迭代的平均离散距离。计算属性数组的离散距离，仅需关注当前访问地址 a_i 与前一次访问地址 a_{i-1} 之间的差值。结合该属性数组在当前迭代中的访问次数 c_i 以及前一次访问结束时的平均离散距离 β_{i-1}，通过式(1)，我们能够计算出 kernel 在当前迭代中访问该属性数组的平均离散距离 β_i。式(1)无需额外存储与每个地址相关的元数据，显著降低了计算复杂度和资源消耗。

$$\beta_i = (\beta_{i-1}(c_i - 1) + (a_i - a_{i-1}))/c_i。 \quad (1)$$

平均离散距离增大时，表明对该属性数组的访问模式趋于离散，即空间局部性减弱。在此情境下，采用较大的缓存行粒度效果有限。因为较大的缓存

行虽能涵盖更多元素，却也延长了 DRAM 访问周期，过大的缓存行使得缓存未命中的开销更大。如果缓存行内仅有部分数据被实际访问，会浪费内存带宽。若 kernel 访问某属性数组时表现出较差的空间局部性，则采用较小的缓存行粒度更为适宜，小缓存行粒度提供了更高的灵活性。每次访存小缓存行都能更精确地读取实际需求的数据，从而缩短访存时间，减轻缓存未命中的负面影响，并提升内存带宽的有效利用率。

为了应对图计算中多样化的空间局部性，AB-Cache 采用了灵活的自适应策略。该策略通过设立 2 种不同配置的缓存结构 A-Cache 与 B-Cache，分别缓存不同空间局部性的访存请求。A-Cache 专为那些平均离散距离较大、空间局部性较弱的属性数组设计，其缓存行粒度被设定为 32 B。B-Cache 则承担着存储剩余属性数组、边数组、顶点数组的职责，其缓存行粒度扩展至 128 B，以更好地利用较强空间局部性。图 2 直观展示了 AB-Cache 的具体结构。A-Cache 与 B-Cache 作为 2 个相互独立的实体，允许系统根据访存请求的特性选择性地访问。A-Cache 与 B-Cache 均采用了广泛使用的组相联结构。

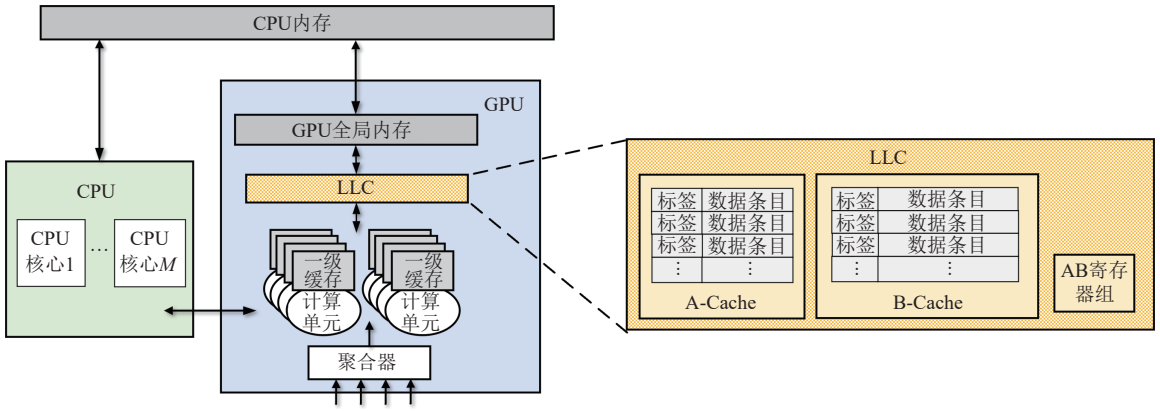


Fig. 2 AB-Cache structure

图 2 AB-Cache 结构

图计算在多数情况下采用迭代方式执行，需要经历多次迭代，直至达到收敛状态或完成对图数据的处理。GPU 平台上的图计算应用通常包含多个 kernel 函数。当 GPU 执行这些 kernel，会因 kernel 的切换或同一 kernel 的多次迭代而触发上下文切换。模拟器利用 GPU 提供的接口可以追踪并识别图计算每一轮迭代实际运行的 kernel。在模拟环境中识别图计算不同属性数组的内存地址边界是一项相对直观的任务。在初始化阶段，属性数组的内存分配通过函数 *cudaMalloc* 完成，该函数由 CUDA 运行时[32]提供。GPGPU-Sim 模拟器[33]重新实现了包括函数 *cudaMalloc* 在内的多个 CUDA 运行时函数。通过监听和解析函数 *cudaMalloc* 的返回值，我们能够获取每个属性数组的内存地址边界。

不同的图计算应用包含不同数量的 kernel 和不同数量的属性数组。我们首先统计单个 kernel 访问每个属性数组时的空间局部性。将统计范围从单个 kernel 扩展到所有 kernel，分别统计每个 kernel 访问多个属性数组时的空间局部性。在本文中，我们手动计算 kernel 访问属性数组时的平均离散距离，并以此作为评估空间局部性的关键指标。为了更直观地比较 kernel 访问不同属性数组时空间局部性的差异，我们将平均离散距离转化为相对值，即相对平均离散距离。该值以百分比形式表示，反映了每轮迭代中 kernel 访问不同属性数组时平均离散距离的相对比例。

对于每个 kernel，我们依据式(2)确定哪些属性数组应当被 A-Cache 缓存。对于访问频次极低的属性数组，对应的相对平均离散距离因样本量不足而缺乏参考价值，因此需要从分析中剔除。我们定义 kernel 访问属性数组 k 时的相对平均离散距离为 P_k，并将 kernel 访问所有属性数组产生的相对平均离散距离记为集合 P，同时定义属性数组的全集为 N，而最终被选定缓存至 A-Cache 的属性数组子集为 t。我们采用方差分析来评估不同属性数组间空间局部性的相似度。当所有属性数组的相对平均离散距离的方差小于阈值 γ 时，我们认为这些访存请求的空间局部性较为一致，将所有属性数组均缓存至 A-Cache。若方差大于阈值 γ，则表明不同访存请求间的空间局部性存在显著差异，或个别访存请求的空间局部性显著偏离整体。我们仅将空间局部性最差的访存请求所对应的属性数组缓存至 A-Cache。我们将阈值 γ

设置为 0.01。

$$t=\begin{cases}N, \text{ if } \sigma^2(P)\leqslant\gamma,\\ \{k\}, \text{ if } \sigma^2(P)>\gamma \text{ and } \max(P)=P_k,\ k\in N。\end{cases} \tag{2}$$

至此，可以在执行图计算的某个 kernel 时，确定哪些属性数组应该缓存到 A-Cache。为了降低 AB-Cache 的复杂度，我们聚合了属于同一图计算应用的所有 kernel 的决策。以 k-Core 为例，若原本 kernel 1 选择将属性数组 1 缓存至 A-Cache，而 kernel 2 则选择属性数组 0，通过聚合后，二者共同决定了在每轮迭代中，属性数组 0 和 1 都将被缓存至 A-Cache。此时，无需根据 kernel 选择缓存到 A-Cache 的属性数组，因为图计算所有迭代中需要被缓存到 A-Cache 的属性数组是一致的。AB-Cache 结构简洁高效，对操作系统和应用程序透明，无需额外修改操作系统和程序代码。AB-Cache 可以很容易地集成到各种现有的图加速器中，以最小的硬件成本获得较好的加速效果。

3 在线自动分类

3.1 访存请求分类

本文基于相对平均离散距离实现了访存请求在线自动分类。相比第 2 节的手动分类，我们进一步完善了基于自适应策略的 AB-Cache 的工作流程。每轮迭代完成后，追踪并记录 kernel 访问每个属性数组的相对平均离散距离。每轮迭代结束时会重置访问次数 c 和平均离散距离 β。GPU 执行的图计算应用由多个 kernel 组成时，这些 kernel 会以交替方式执行，仅在每个 kernel 实际执行期间收集该 kernel 相关的访存数据。除了少数仅涉及初始化操作的 kernel 外，图计算的每个 kernel 通常都会经历多轮迭代，生成多组关于相对平均离散距离的数据集。图 3 包含多组图计算应用和图数据的组合，每个子图展示 kernel 访问属性数组时相对平均离散距离的变化趋势，子图中的每条曲线代表特定属性数组的相对平均离散距离的趋势。

广度优先搜索(breadth first search, BFS)算法仅涉及单个属性数组，因此无需进行相对平均离散距离的统计。观察 Graph-Coloring(GC)算法，kernel 在访问 2 个属性数组时展现出一致的相对平均离散距离，且这一特征在 kernel 执行过程中呈现出稳定的趋势。单源最短路径(single-source shortest paths, SSSP)算法更为复杂，包含 2 个 kernel，每个 kernel 均涉及 3 个属性数组的访问。对于每个 kernel，尽管空间局部性随迭代的变化不如 GC 算法那般平稳，但依然能够

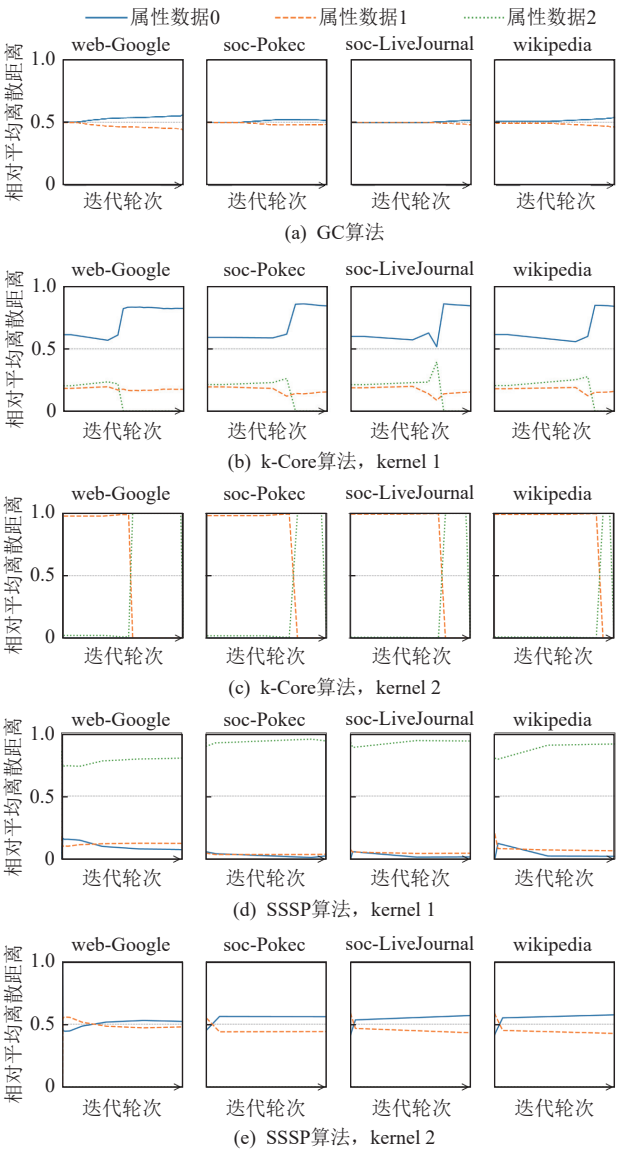


Fig. 3 Trend of relative average discrete distance when the kernel accesses the property array during the graph computing

图 3 图计算中 kernel 访问属性数组的相对平均离散距离趋势

清晰地区分出具有较高相对平均离散距离的属性数组。k-Core 算法的 kernel 2 空间局部性特征与 SSSP 算法的 2 个 kernel 相似。k-Core 算法的 kernel 1 在执行过程中，对属性数组的访问出现了突变。kernel 1 访问属性数组 1 时的相对平均离散距离显著高于访问属性数组 2 时，这一趋势占据了 kernel 1 执行周期的大部分时间。

图 3 显示，图计算执行过程中，每个 kernel 访问属性数组的行为在多个迭代间呈现出较强的持续性。我们仅分析并记录每个 kernel 在其首轮执行时的访问行为模式，并预测该 kernel 在后续所有迭代中都将

遵循和首轮迭代类似的模式。此判定过程仅需在图计算执行初期执行 1 次。通过在线自动分类方案，我们将手动分类属性数组转换为自动分类。自动分类方案在 GPU 的运算核心内实现。针对每个属性数组，分配 5 个寄存器：2 个寄存器用于追踪并计算平均离散距离（其中一个记录访问的内存地址 a，另一个记录计算得到的平均离散距离 β），1 个寄存器用来累计属性数组的访问次数 c。平均离散距离的计算过程被设计为可与 DRAM 访问操作并行执行。每个属性数组还额外配置了 2 个 AB 寄存器组 ABR（array boundary registers），用于存储属性数组的边界信息。这些 AB 寄存器在特定情况下会被重置为 0，比如属性数组被缓存至 B-Cache 或分类过程尚未完成。我们将其放在 AB-Cache 旁边而不是运算单元中，降低内存控制器的访问延迟。

3.2 AB-Cache 工作流程

结合自动分类算法，AB-Cache 的工作流程如下：

访问 AB-Cache 时，存在 2 种情况。若当前为首次迭代，AB-Cache 同时查询 A-Cache 和 B-Cache；若任一子 Cache 命中，则视为 AB-Cache 命中；若二者均未能命中，则认为 AB-Cache 未命中。如果自动分类已完成，则根据 AB 寄存器的状态，决定查询 A-Cache 还是 B-Cache。如果 AB-Cache 未命中，则相应的访存请求会被发送至内存控制器。当内存控制器从 DRAM 中获取数据时，根据 AB 寄存器的指示，这些数据会被定向缓存至 A-Cache 或 B-Cache 中。A-Cache 与一级缓存采用不同的缓存行粒度。为了简化 AB-Cache 的复杂度，当数据被写入 A-Cache 时，这些数据将不会同步缓存至一级缓存。在分类操作执行前，相关数据已被送入处理器，从而有效隐藏了分类过程引入的延迟。

4 实验结果及分析

4.1 实验配置

本文使用 GPGPU-Sim 模拟器平台[33]验证 AB-Cache 架构的有效性，参考 NVIDIA A100 GPU[34]的具体架构特性进行建模。为了优化模拟效率，我们设定了模拟时间上限。若算法执行时间超过阈值，模拟将自动终止。表 2 展示了模拟器的基础配置，通过修改 LLC 和计算单元的代码将 AB-Cache 结构添加到模拟器。本文采用 GPU 原有的二级缓存作为性能评估的基准，并与 RRIP 策略及 AB-Cache 架构进行对比。RRIP 策略旨在通过精确识别数据重用距离，挖掘并利用更为细致的数据局部性特性。为此，本文还将 RRIP 移植到 GPU 架构中。

本文从 GraphBIG 算法套件[14]中挑选了 5 个图计算应用，用来评估不同优化策略在 GPU 平台上的性能。k-Core 算法聚焦于挖掘图中的紧密子图，通过设定顶点度数阈值，高效识别并提取符合条件的子图结构；GC 算法专注于解决图的 M 可着色问题，核心在于判断给定图和颜色集合是否能满足所有相邻顶点颜色各异的条件；Betweenness-Centr（BC）算法分析图中顶点在信息流或资源传递中的关键作用，通过统计每个顶点作为其他任意 2 个顶点间最短路径桥梁的次数来量化中心性指标。BFS 算法作为图计算领域的基础算法，为众多高级图计算应用提供了原型框架；SSSP 算法专注于解决图论中的经典挑战，即快速计算出从指定源点到图中所有其他顶点的最短路径长度。

为了探究 AB-Cache 结构对非图计算应用的影响，我们选取了 4 个来自不同领域的 GPU 基准测试程序[35]：库仑势（Coulomb potential，CP）[36]、MUM（MUMmer-GPU）、N 皇后（N-queen，NQU）、神经网络（neural network，NN）。本文对比这些基准测试在 GPU 集成 AB-Cache 结构前后的性能。CP 作为 Parboil[37]基准套件的一部分，专注于分子动力学中的 CP 计算，是物理学领域众多问题求解的基石。MUM 则是一款高效的 DNA 精确序列比对工具，其 GPU 加速版相较于传统的 CPU 版（MUMmer），实现了超过 3 倍的速度提升。NQU 算法利用 CUDA 库在 NVIDIA GPU 上并行求解 NQU 问题，通过并行搜索策略显著加速了求解过程。最后，NN 算法在 GPU 上部署了一个神经网络模型，充分利用了 GPU 在向量和矩阵运算上的并行优势，这些运算在 GPU 的并行架构下能够高效执行。

本文采用表 3 所列的多样化图数据集作为输入[15]，所有图的大小都超过了 LLC 容量。这些图在数据规模及拓扑结构方面展现出显著差异，顶点与边的数量跨越了较大的数量级范畴，为全面验证 AB-Cache 结构的性能提供了丰富的测试基础。为便于表述，

Table 3 Graph Dataset

表 3 图数据集

图	顶点数量	边数量	大小/MB	平均度数
WG	0.88×10^6	5.10×10^6	70.30	11
SP	1.63×10^6	30.62×10^6	424	18
SLJ	4.85×10^6	68.99×10^6	1 011.61	14
WP	3.57×10^6	45.03×10^6	642.70	12

本文对所用图的名称进行了如下缩写：web-Google 缩写为 WG，soc-Pokec 缩写为 SP，soc-LiveJournal 缩写为 SLJ，wikipedia 缩写为 WP。

4.2 性能加速比

本节汇总了不同优化策略的平均加速比，并通过图 4 直观地展示了实验结果。我们将基线方案的性能设为标准参考点。与基线相比，RRIP 方案的平均加速比为 0.99，AB-Cache 结构的平均加速比为 1.14。与 RRIP 方案对比，AB-Cache 仍能带来 1.14 的平均加速比优势。

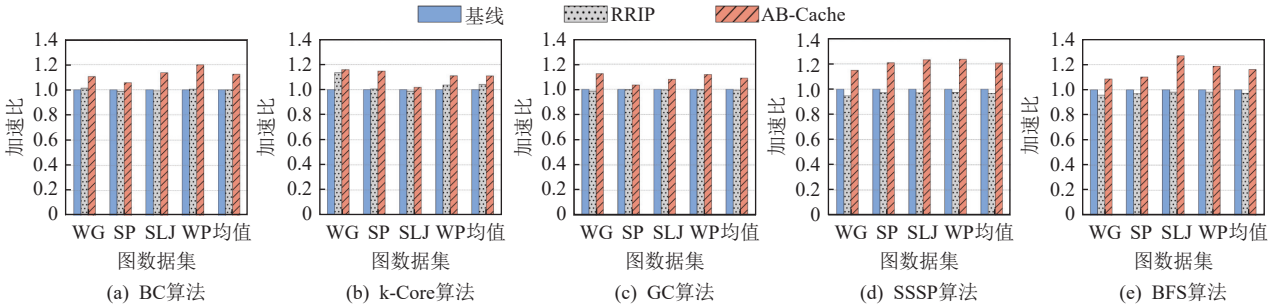


Fig. 4 Speedup obtained by RRIP and AB-Cache compared with the baseline

图 4 与基线相比，RRIP 和 AB-Cache 获得的加速比

基线与 RRIP 方案都使用单一缓存结构处理具有不同空间局部性的访存请求。由于缺乏针对性，这 2 种方案的性能并不理想。此外，这 2 个方案均使用 128 B 的缓存行粒度，在面对空间局部性较差的访存请求时，会浪费宝贵的缓存容量和内存带宽。基线方案较低的缓存命中率导致大量不必要的 DRAM 访问。RRIP 方案试图通过细化数据重用距离来提升性能，但实际应用中效果有限，仅在 k-Core 算法处理 web-Google 图数据时表现出一定的加速效果。其他情况下 RRIP 方案甚至因错误评估数据重用性，保留了低重用性数据而不是高重用性数据，反而导致性能下降。

AB-Cache 采用自适应的思想，巧妙融合了 2 种不同的缓存设计。具体地，A-Cache 采用较小的 32 B 缓存行，在面对空间局部性较差的访存请求时能够显著降低 DRAM 访问延迟。另一方面，B-Cache 则配备 128 B 的缓存行，旨在最大限度地利用空间局部性提升缓存效率。当 kernel 涉及顶点数组和边数组的访问时，这些数据结构展现出比属性数组更强的空间局部性和访存规律性。此时，B-Cache 的大缓存行设计能够更有效地捕获并利用这些局部性特征，从而显著提升缓存命中率。

4.3 AB-Cache 对非图计算应用性能的影响

本节探究 AB-Cache 对非图计算应用的影响。我们选取了 4 个在 GPU 中广泛应用的基准测试程序：CP，MUM，NQU，NN。A-Cache 与 B-Cache 可以看作 AB-Cache 的 2 个独立 bank。我们摒弃了基于空间局部性对内存访问请求进行分类的方法。由于 A-Cache 中的数据不会被缓存到一级缓存，这导致一级缓存的命中率略微下降。因此，需要适当降低 A-Cache 的容量，减少对一级缓存的影响。如果大幅度降低 A-Cache 的容量，导致 B-Cache 的容量占缓存总容量的绝大部分，AB-Cache 会退化为传统的组相联 Cache，使用这类缓存并不能测试 AB-Cache 结构对非图计算应用的性能影响。为了最小化对性能的干扰，我们根据预设的比例(1∶2)，将内存访问请求分配给 A-Cache 和 B-Cache。

与基线相比，AB-Cache 实现了平均 1.03 倍的性能加速，实验结果在图 5 中详尽展示。与图计算应用相比，这些非图计算应用的访存请求的空间局部性差异并不大。因此，B-Cache 和 A-Cache 都能获得令人满意的性能。AB-Cache 架构不会对非图计算应用的性能造成负面影响，在某些情况下还能实现轻微的性能提升。相比基线，在 GPU 体系结构中包含

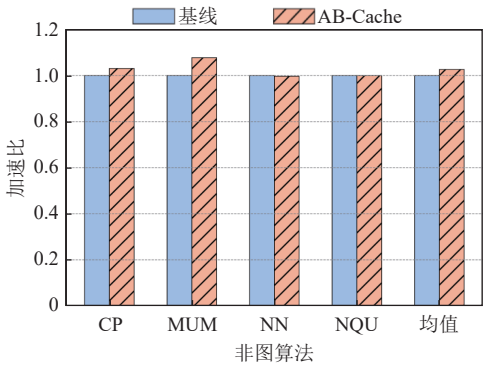


Fig. 5 Performance of non-graph computing applications under AB-Cache structure

图 5 AB-Cache 结构下非图计算应用的性能

多种缓存行粒度可以更具针对性地优化程序性能。

4.4 AB-Cache 性能分析

本节探讨了 BC 算法在处理 web-Google 时的内存访问特性。经过充分的程序预热后，我们聚焦于第 22 轮迭代，揭示图计算包含的不同空间局部性。实验结果见图 6，横轴代表当前迭代的时钟周期，纵轴代表地址空间。图 6 中的每个点都表明在一个特定时钟周期对某一具体地址的访问。紧挨横坐标的 4 个虚线区间内的点表示对 4 个属性数组的访问，第 5 个虚线区间内的点表示对顶点数组的访问，距离横坐标最远的虚线区间内的点代表对边数组的访问。

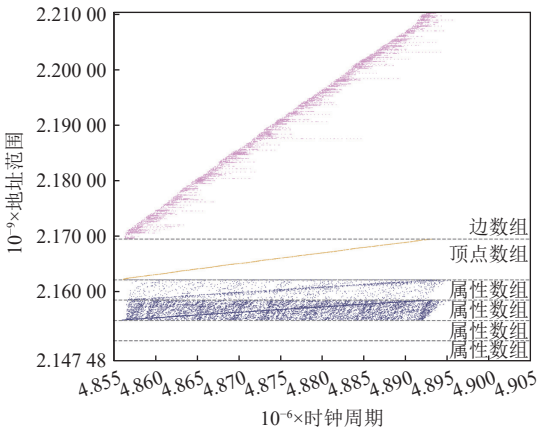


Fig. 6 Locality exploration in a single iteration of graph computing

图 6 图计算单次迭代中的局部性探索

BC 算法显现出 2 种不同的空间局部性特征。在当前迭代，算法对第 1、第 2 个属性数组的访问完全缺席，对第 3、第 4 个属性数组的访问则呈现一种不规律的、离散的点状模式。这些离散点之间缺乏紧密联系，表明访问这 2 个属性数组时空间局部性较差。当算法访问边数组和顶点数组时，访存行为转变为有序的线性模式，表明这些访存地址之间具有良好的连续性。这种高度可预测的访问行为是优秀空间局部性的体现。此外，边数组的一些数据在短时间内被频繁重复访问，展现出一定的时间局部性。AB-Cache 采用自适应思想，巧妙融合了 2 种不同缓存设计，可以同时处理 2 种不同的空间局部性。与之相对，传统缓存优化方案使用单一缓存结构，对同时存在的 2 种空间局部性都没有针对性优化，性能自然不如 AB-Cache。

为了进一步解释 AB-Cache 获得的性能提升，本文探究了图计算整个执行过程的空间局部性特征，并通过图 7 的百分比堆积柱状图展示了实验结果。

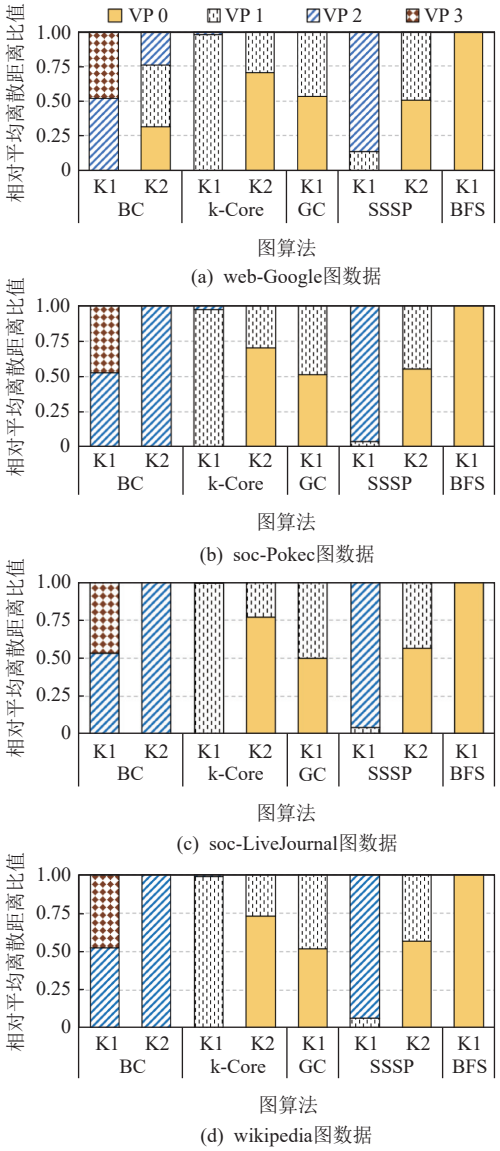


Fig. 7 Relative average discrete distance of each kernel accessing the property arrays

图 7 每个 kernel 访问属性数组的相对平均离散距离

图中的每根柱子表示 kernel #访问不同属性数组时产生的多个相对平均离散距离的比值。kernel 编号缩写为 K #，属性数组编号缩写为 VP #。除了 BFS 和 GC，其余图计算应用均包含至少 2 个 kernel，且每个算法内不同 kernel 访问属性数组时，展现出不同的空间局部性。每个 kernel 访问不同的属性数组时行为各不相同，这种差异直接导致属性数组间相对平均离散距离的显著不同。以 k-Core 算法处理 web-Google 为例。kernel 1 对属性数组 0 的访问频次极低，因此其相对平均离散距离缺乏足够的参考价值。对于属性数组 1 和属性数组 2，尽管 kernel 1 对二者的访问次数相近，但属性数组 1 的相对平均离散距离却远高于属性数组 2，这一现象明确指出 kernel 1 访问属

性数组 1 时具有更差的空间局部性。

图 6 表明图计算每轮迭代存在不同类型的空间局部性。图 7 表明图计算整个执行过程与每轮迭代类似，也存在不同类型的空间局部性。AB-Cache 采用灵活的策略，通过配置不同的缓存结构来适配不同的空间局部性。这一创新设计使得 AB-Cache 在与其他方案的对比中，展现出了更为卓越的性能提升效果。

4.5 缓存命中率分析

图 8 展示了执行 k-Core 算法时，不同方案对缓存命中率的提升。与基线和 RRIP 策略相比，AB-Cache 按照自适应思想，为具有不同空间局部性的内存访问请求自适应选择合适的缓存行粒度。这一设计显著提升了二级缓存的命中率。A-Cache 采用 32 B 缓存行粒度，大幅降低了内存访问延迟。在这 2 个关键因素的共同作用下，AB-Cache 相较于基线配置实现了平均 1.14 倍的性能提升。由于 A-Cache 与一级缓存的缓存行粒度不同，A-Cache 中的数据无法被缓存至一级缓存。这一设计选择导致相较于基线而言，

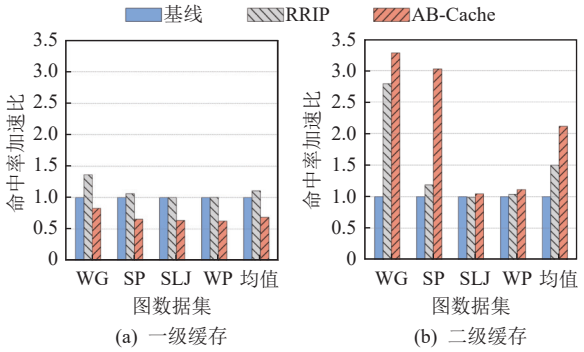


Fig. 8 Effect of AB-Cache and RRIP on cache hit rates

图 8 AB-Cache 和 RRIP 对缓存命中率的影响

AB-Cache 架构下一级缓存命中率下降。

图 9 为关于 A-Cache 与 B-Cache 命中率的占比堆积柱状图。除了 BFS 算法处理 soc-Pokec 图，其余场景 A-Cache 的命中率均超越 B-Cache。A-Cache 与 B-Cache 的容量相同，但实验数据表明 AB-Cache 架构中 A-Cache 在性能提升方面扮演着更为关键的角色。相较于仅依赖单一缓存结构的基线和 RRIP 方案，采用自适应思想的 AB-Cache 结构展现出更为优越的性能。

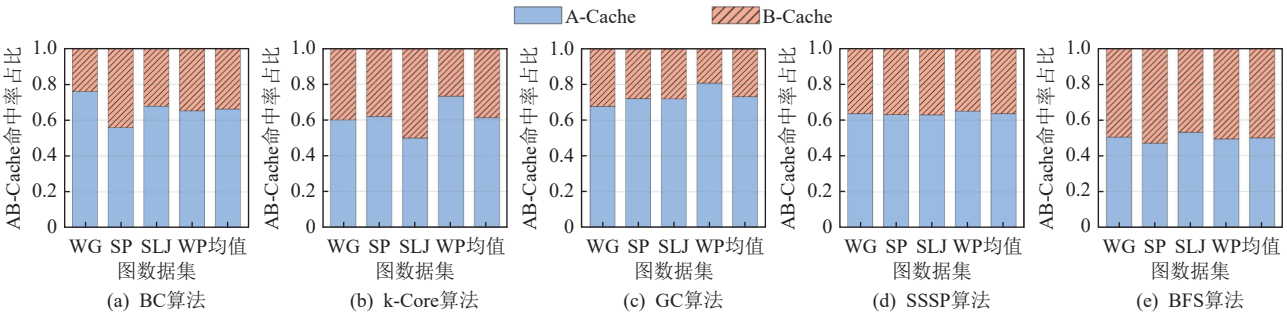


Fig. 9 Comparison of hit rate between A-Cache and B-Cache in AB-Cache

图 9 AB-Cache 中 A-Cache 和 B-Cache 命中率比较

4.6 面积与功耗

在 22 nm 工艺下，我们借助 CACTI 7.0[38] 工具对 AB-Cache 与基线方案进行建模，评估了它们的面积、能耗、访问时间。表 2 详细列出了这 2 种设计方案的参数，表 4 汇总了实验结果。AB-Cache 结构的面积、能耗、访问时间分别为基线方案的 1.30 倍、0.74 倍、0.86 倍。A-Cache 与 B-Cache 是 2 个相互独立的缓存。A-Cache 与 B-Cache 均被设计为基线缓存容量的一半。B-Cache 的容量比基线缓存少一半，但外围电路面积并未同比例缩减，使得 B-Cache 的总面积占基线缓存的 61.11%。A-Cache 包含更多的标签条目，占用面积达到基线缓存的 69.22%。单独访问 A-Cache 或 B-Cache，延迟与功耗都优于基线缓存。在线自动分类算法完成分类后，需要访问 LLC 时，实际只访问 A-Cache 或 B-Cache 中的 1 个。与基线缓存相比，AB-Cache 在访问延迟和功耗方面表现出更好的性能。

Table 4 Area, Energy and Access Time of AB-Cache and Baseline Cache

表 4 AB-Cache 和基线缓存的面积、能耗、访问时间

缓存	面积/mm^2	每次访问能量/nJ	访问时间/ns
AB-Cache	49.35	0.67	5.15
基线缓存	37.86	0.91	5.96

本文扩展了基线缓存的容量，直至其面积与 AB-Cache 相等，扩展后的基线缓存标记为基线缓存 Ⅱ。与原始基线缓存相比，基线缓存 Ⅱ 和 AB-Cache 分别获得了 1.09 倍和 1.14 倍的性能提升。这一发现表明，单纯增加缓存容量，并不能使传统缓存架构达到与 AB-Cache 相媲美的性能优化效果。

4.7 与其他方案的比较

为了验证 AB-Cache 的先进性，本文将其与多个方案进行对比。本节首先将 AB-Cache 与 CPU 平台的图计算优化方案 GRASP[12]，DROPLET[39]，GraphFire[40] 进行对比。相比基线缓存方案，GRASP，DROPLET，GraphFire，AB-Cache 分别获得了 0.20 倍、0.27 倍、0.28 倍、1.21 倍的加速比，实验结果如图 10 所示。实验结果表明，GPU 平台上 AB-Cache 获得的加速比远远超过其他方案。

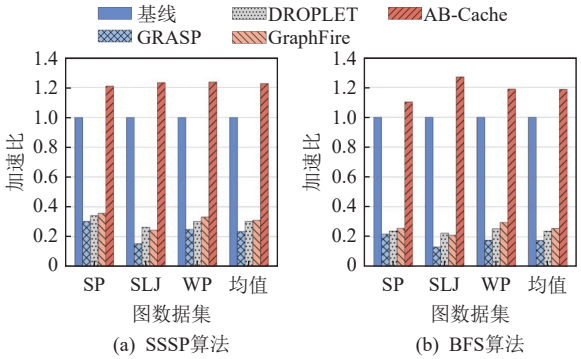


Fig. 10 Speedups obtained by GRASP, DROPLET, GraphFire, and AB-Cache compared with baseline cache

图 10 GRASP，DROPLET，GraphFire，AB-Cache 相比基线缓存获得的加速比

图计算是一种典型的访存受限应用，其计算访存比较低。相比 CPU 平台，采用 SIMT 执行模型的 GPU 平台更适合加速图计算的执行。GPU 平台对图计算的优化工作大多专注于图计算执行流程或图计算框架，而不只优化 GPU 缓存，这与 AB-Cache 只优化缓存的思路不同。因此，这些方案很难与 AB-Cache 进行比较。本文将 GRASP 优化方案从 CPU 平台移植到 GPU 平台，记为 GRASP-GPU。本文以图 SLJ 作为输入数据执行算法 BC 和 SSSP，测试 GRASP-GPU 的性能。实验结果表明，与基线缓存相比，GRASP-GPU 和 AB-Cache 分别获得了 1.06 和 1.13 的加速比。GRASP-GPU 获得加速的原理是将高度数顶点和低度数顶点隔离，并使用高效的缓存管理策略让高度数顶点尽可能保留在缓存中。GRASP 方案没有针对不同重用性的数据实施不一样的管理策略，这是导致 GRASP-GPU 的性能不如 AB-Cache 的重要原因。

4.8 敏感性分析

1）A-Cache 与 B-Cache 容量比例对性能的影响

本节分析 A-Cache 与 B-Cache 设置不同的容量比例时系统性能的变化。我们将 A-Cache 和 B-Cache 的容量设置为 1∶2 和 2∶1，分别记为 AB-Cache_12 和 AB-Cache_21。本节使用的算法为 BFS 和 SSSP，使用的图数据集为 WG，SP，SLJ，WP。AB-Cache 的总容量依然为 40 MB。实验结果如图 11 所示，相比基线缓存，AB-Cache，AB-Cache_12，AB-Cache_21 分别获得 1.18，1.06，1.17 的加速比。

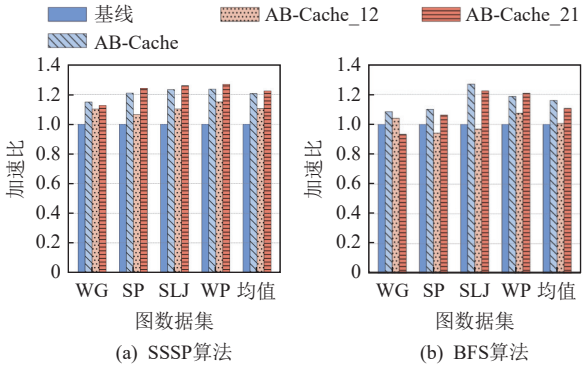


Fig. 11 Impact of the capacity ratio of A-Cache and B-Cache on performance

图 11 A-Cache 与 B-Cache 容量比例对性能的影响

与 1∶1 配置相比，当 A-Cache 与 B-Cache 的容量比例设为 1∶2 或 2∶1 时，系统性能均有所下降。本节使用 GPGPU-Sim 模拟器输出图计算执行时的访存地址流，并使用 Python 程序进行数据分析。统计结果显示，绝大多数访存地址集中在边数组和属性数组，而顶点数组的访问次数占比较低；边数组与属性数组的访问次数占比约为 1∶1。实验结果如图 12 所示。

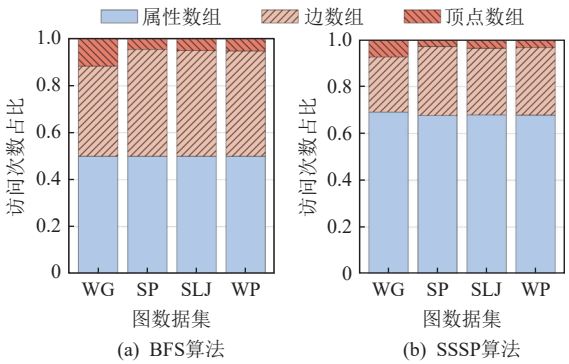


Fig. 12 Access frequency distribution across different arrays in the memory access address streams of graph computing

图 12 图计算访存地址流中不同数组的访问频率分布

根据图 12 的实验结果，A-Cache 和 B-Cache 的容量占比为 1∶2 或 2∶1 时，2 个缓存的负载并不均衡，分别导致 A-Cache 或 B-Cache 负载过高，缓存命中率下降，进而影响整体性能。在 SSSP 算法中，处理图 SP，SLJ，WP 时 AB-Cache_21 比 AB-Cache 的性能更高。采用更大的 A-Cache 能够进一步提高图计算中不规

则访存地址流在 AB-Cache 的命中率，这能够抵消 A-Cache 中的数据不能缓存到一级缓存的负面影响。本文提出的 AB-Cache 结构需在不损害非图计算应用性能的前提下提升图计算性能，因此将 A-Cache 与 B-Cache 的容量占比设为 1∶1 是一种合理的选择。

2）阈值γ对性能的影响

本文采用方差分析来评估不同属性数组间空间局部性的相似度。当所有属性数组的相对平均离散距离的方差小于阈值γ时，将所有属性数组均缓存至 A-Cache。阈值γ默认设置为 0.01，是一个经验值。为了验证不同的γ值对性能的影响，本节增加了敏感性实验，验证γ值在 0.000 01~1 范围内 AB-Cache 的性能变化。本节使用的算法为 k-Core 和 GC，使用的图数据为 WG，SP，SLJ，WP。实验结果如图 13 所示。

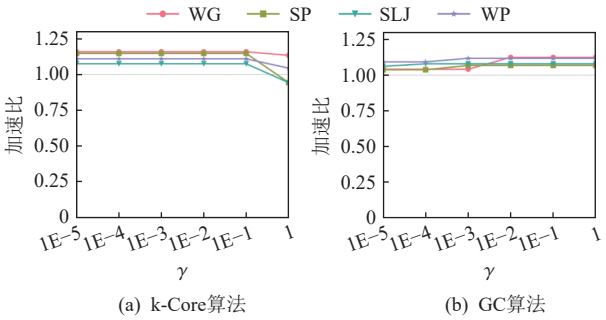


Fig. 13 Effect of γ on the performance of AB-Cache

图 13 γ值对 AB-Cache 性能的影响

AB-Cache 对γ值的敏感性并不高。在一定范围内，改变γ值并不会显著影响总体性能。γ值过大时，对地址流的筛选作用变弱，过多属性数组被缓存到 A-Cache。极端情况下所有属性数组被缓存到 A-Cache，导致本文方案失去在线自动分类效果，回退到最原始的方案：直接将所有属性数组缓存到 A-Cache。如图 13 中 k-Core 算法，当γ=1 时，相比基线缓存可以获得 1.02 加速比。γ值过小会导致过度分类，2 个相对平均离散距离近似的属性数组本应该都缓存到 A-Cache，过小的γ值导致 1 个属性数组被缓存到 B-Cache，导致性能下降。如图 13 中 GC 算法，γ=0.000 01 时，相比基线缓存可以获得 1.06 加速比。

5 相关工作

提升缓存性能的方法之一是优化替换策略[1-2,4]。传统的 LRU（least recently used）[41] 策略通过移除最长时间未被访问的缓存行来优化性能。RRIP[18] 作为 LRU 的进阶版本，巧妙地为每个缓存行引入 2 b 的记录机制，显著增强了性能，且能无缝集成至现代处理器的近似 LRU 实现中。SDBP（sampling dead block prediction）[26] 与 Leeway[29] 策略通过精确预测不再使用的缓存块，进一步提升了缓存性能。Hawkeye[22-23] 与 SHiP[21] 替换算法则以其卓越的性能超越了众多传统方法。XMem（expressive memory）[42] 将高频访问的数据固定于缓存中，减少因频繁替换造成的抖动。这一做法牺牲了缓存的灵活性，同时忽视了低频访问数据的局部性。GRASP[12] 是在最后一级缓存进行领域专用缓存管理的优化方案。GRASP 使用软件重排序技术将高度数顶点聚集在一起，从而隔离高度数和低度数顶点，通过保护热顶点不受缓存抖动的影响来最大化热顶点的重用，同时保持足够的灵活性来根据需要捕获其他顶点的重用。GRASP 利用轻量级软件支持精确定位热顶点，使硬件成本可以忽略不计，避免了其他缓存管理方案使用的存储密集型预测机制带来的额外开销。DROPLET[39] 是一种用于图计算的数据感知解耦的预取器，作用于最后一级缓存。DROPLET 根据数据固有的重用距离对不同的图数据类型进行不同的预取策略。DROPLET 在物理上解耦，以克服由于不同数据类型之间的依赖链而导致的序列化。GraphFire[40] 通过识别图计算中不规则的访问模式，利用定制的访问粒度、数据感知的插入、基于频率的替换来解决间接访问带来的问题，而不依赖软件或 ISA 支持。尽管这些策略依赖于复杂的预测算法，并且实现难度比 AB-Cache 更大，但它们无疑为缓存优化领域开辟了新的探索路径。

存在一些使用混合缓存结构提高图计算或图挖掘性能的方案。GRAMER[43] 包含一个专用的缓存层次结构，高价值的数据（通过成本有效的启发式方法精确识别）永久驻留在高优先级缓存中，而其他数据则保存在低优先级缓存中，这和 Xmem 方案的基本思想类似。GRAMER 为低优先级缓存设计了轻量级替换策略。此外，GRAMER 还设计了流水线处理单元，最大限度地提高了计算的并行性。GRAMER 通过预处理获得顶点的优先级，AB-Cache 则通过收集运行时信息在线判定数据映射到哪个缓存结构。AB-Cache 只需简单修改即可扩展到处理动态图的场景。GRAMER 和 AB-Cache 都采用混合缓存架构，但二者的目的不同。GRAMER 将缓存拆分为高优先级和低优先级 2 部分，通过区分顶点的优先级并设计新的替换策略提高性能。AB-Cache 则在区分重用性的基础上进一步针对不同空间局部性设计了不同的缓存结构。

SCC(slanted cache compression)[44]与 YACC(yet another compression cache)[45]取得了显著效果。这 2 种方法均采取分层压缩策略，首先独立压缩每个缓存行，再将连续压缩的缓存行合并，以实现更高效的存储空间利用。基于字典的 DISH 方案[46]将缓存行分割成多个子块，并对这些子块进行编码，同时利用共享字典机制减少存储开销。然而，在图计算场景中，由于频繁的缓存替换，需要频繁执行压缩和解压缩操作。这些基于压缩的方案面临挑战[47]。

图计算框架致力于加速开发过程并提升开发效率[48-49]。当前的一个核心焦点在于构建高效的共享内存图计算框架[8,50-52]，旨在通过优化内存使用提升图计算的性能。与此同时，随着大数据时代的到来，分布式图计算框架[53]也应运而生。这些框架专为处理超大规模的图数据而设计，能够有效应对数据量的爆炸性增长。面对主存容量的限制，核外图计算框架[54-56]成为一种重要的解决方案。

SRAM 存内处理技术的兴起为加速图计算提供了新的机会[57]。使用存内计算能够克服图计算中的内存墙问题。目前，AB-Cache 结构部署在单计算机中，集成到其他图计算框架或其他研究方案是未来研究的一个重要方向。

6 结　论

本文分析了在 GPU 平台执行图计算时存储系统面临的挑战。通过为图计算中多样化的局部性特征定制不同的优化策略，能够提升图计算的整体性能。本文提出了面向 GPU 平台的 AB-Cache 架构，该架构采用自适应的思想，使用不同缓存结构优化具有不同空间局部性特征的内存访问请求。AB-Cache 不仅保持了结构简单与高效的特点，还对操作系统及应用程序完全透明，可轻松集成至各类现有的图计算加速器中。

作者贡献声明：贾朝阳提出了架构思路和实验方案，并且完成实验和撰写论文；陈湜参与了架构的设计；刘京雨协助修改论文；沈立提出指导意见并修改论文。

参　考　文　献

[1] Easley D, Kleinberg J. Networks, Crowds, and Markets: Reasoning About a Highly Connected World [M]. Cambridge, UK: Cambridge University Press, 2010

[2] Bui T D, Ravi S, Ramavajjala V. Neural graph learning: Training neural networks using graphs [C] // Proc of the 11th ACM Int Conf on Web Search and Data Mining. New York: ACM, 2018: 64–71

[3] Su Hongyu, Wei Xiong, Ye Wu, et al. Quatcnnex: A knowledge graph embedding model based on quaternion convolutional neural network [C] // Proc of the 3rd Int Conf on Big Data, Artificial Intelligence and Internet of Things Engineering. Piscataway, NJ: IEEE, 2022: 320–326

[4] Zeng Xiangxiang, Song Xiang, Ma Tengfei, et al. Repurpose open data to discover therapeutics for COVID-19 using deep learning[J]. Journal of Proteome Research, 2020, 19(11): 4624−4636

[5] Balaji V, Crago N, Jaleel A, et al. P-OPT: Practical optimal cache replacement for graph analytics [C] // Proc of the 27th IEEE Int Symp on High-Performance Computer Architecture. Piscataway, NJ: IEEE, 2021: 668–681

[6] Gonzalez J E, Low Yucheng, Gu Haijie, et al. PowerGraph: Distributed graph-parallel computation on natural graphs [C] // Proc of the 10th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2012: 17–30

[7] Pearce R, Gokhale M, Amato N M. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates [C] //Proc of the 27th Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2014: 549–559

[8] Chi Yuze, Dai Guohao, Wang Yu, et al. NxGraph: An efficient graph processing system on a single machine [C] // Proc of the 32nd IEEE Int Conf on Data Engineering. Piscataway, NJ: IEEE, 2016: 409–420

[9] Lü Yashuai, Guo Hui, Huang Libo, et al. GraphPEG: Accelerating graph processing on GPUs[J]. ACM Transactions on Architecture and Code Optimization, 2021, 18(3): 1−24

[10] Yesil S, Heidarshenas A, Morrison A, et al. Speeding up SpMV for power-law graph analytics by enhancing locality & vectorization [C] // Proc of the 33rd Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2020: 1–15

[11] Zhang Dunbo, Jia Chaoyang, Shen Li. Compressed page walk cache[J]. Frontiers of Computer Science, 2022, 16(3): 163104

[12] Faldu P, Diamond J, Grot B. Domain-specialized cache management for graph analytics [C] // Proc of the 26th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2020: 234–248

[13] Shin K, Eliassi-Rad T, Faloutsos C. Corescope: Graph mining using k-core analysis - patterns, anomalies and algorithms [C] // Proc of the 16th IEEE Int Conf on Data Mining. Piscataway, NJ: IEEE, 2016: 469–478

[14] Nai Lifeng, Xia Yinglong, Tanase I G, et al. GraphBIG: Understanding graph computing in the context of industrial solutions [C] // Proc of the 28th Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2015: 1–12

[15] Davis T A, Hu Yifan. The University of Florida sparse matrix collection[J]. ACM Transactions on Mathematical Software, 2011, 38(1): 1−25

[16] Sethumurugan S, Yin Jieming, Sartori J. Designing a cost-effective cache replacement policy using machine learning [C] // Proc of the 27th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2021: 291–303

[17] Zhang Dunbo, Lang Qingjie, Wang Ruoxi, et al. Extension VM: Interleaved data layout in vector memory[J]. ACM Transactions on Architecture and Code Optimization, 2024, 21(1): 1−23

[18] Jaleel A, Theobald K B, Steely S C Jr, et al. High performance cache replacement using re-reference interval prediction (RRIP)[C]// Proc of the 37th Int Symp on Computer Architecture. New York: ACM, 2010: 60–71

[19] Kim J, Teran E, Gratz P V, et al. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy[J]. ACM SIGPLAN Notices, 2017, 52(4): 737−749

[20] Jimenez D A, Teran E. Multiperspective reuse prediction[C]// Proc of the 50th Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2017: 436–448

[21] Wu C J, Jaleel A, Hasenplaugh W, et al. SHiP: Signature-based hit predictor for high performance caching[C]// Proc of the 44th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2011: 430–441

[22] Jain A, Lin C. Back to the future: Leveraging Belady's algorithm for improved cache replacement[C]// Proc of the 43rd ACM/IEEE Annual Int Symp on Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 2016: 78–89

[23] Panwar A, Bansal S, Gopinath K. Hawkeye: Efficient fine-grained OS support for huge pages[C]// Proc of the 24th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2019: 347–360

[24] Shi Zhan, Huang Xiangru, Jain A, et al. Applying deep learning to the cache replacement problem[C]// Proc of the 52nd Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2019: 413−425

[25] Jain A, Lin C. Rethinking Belady's algorithm to accommodate prefetching[C]// Proc of the 45th ACM/IEEE Annual Int Symp on Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 2018: 110–123

[26] Khan S M, Tian Yingying, Jimenez D A. Sampling dead block prediction for last-level caches[C]// Proc of the 43rd Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2010: 175–186

[27] Teran E, Wang Zhe, Jimenez D A. Perceptron learning for reuse prediction[C]// Proc of the 49th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2016: 1–12

[28] Teran E, Tian Yingying, Wang Zhe, et al. Minimal disturbance placement and promotion[C]// Proc of the 22nd IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2016: 201−211

[29] Faldu P, Grot B. Leeway: Addressing variability in dead-block prediction for last-level caches[C]// Proc of the 26th Int Conf on Parallel Architectures and Compilation Techniques. Piscataway, NJ: IEEE, 2017: 180–193

[30] Jabbour S, Mhadhbi N, Radaoui B, et al. Detecting highly overlapping community structure by model-based maximal clique expansion[C]// Proc of the 6th IEEE Int Conf on Big Data. Piscataway, NJ: IEEE, 2018: 1031–1036

[31] Niu Yuyao, Lu Zhengyang, Dong Meichen, et al. TileSpMV: A tiled algorithm for sparse matrix-vector multiplication on GPUs[C]// Proc of the 35th IEEE Int Parallel and Distributed Processing Symp. Piscataway, NJ: IEEE, 2021: 68–78

[32] Garland M, Grand S L, Nickolls J, et al. Parallel computing experiences with CUDA[J]. IEEE Micro, 2008, 28(4): 13−27

[33] Hughes C, Green R, Voskuilen G R, et al. GPGPU-Sim overview[R]. Albuquerque, NM: Sandia National Laboratories, 2019

[34] Choquette J, Gandhi W, Giroux O, et al. NVIDIA A100 Tensor Core GPU: Performance and innovation[J]. IEEE Micro, 2021, 41(2): 29−35

[35] Bakhoda A, Yuan G L, Fung W W L, et al. Analyzing CUDA workloads using a detailed GPU simulator[C]// Proc of the 10th IEEE Int Symp on Performance Analysis of Systems and Software. Piscataway, NJ: IEEE, 2009: 163−174

[36] Ryoo S, Rodrigues C I, Baghsorkhi S S, et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA[C]// Proc of the 13th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2008: 73−82

[37] University of Illinois. Parboil benchmark suite[CP/OL]. [2025-10-11]. https://github.com/abduld/Parboil/tree/master

[38] Muralimanohar N, Balasubramonian R, Jouppi N P. CACTI 6.0: A tool to model large caches[R/OL]. Palo Alto, CA: HP Laboratories, 2009 [2025-10-11]. https://api.semanticscholar.org/CorpusID:14245-854

[39] Basak A, Li Shuangchen, Hu Xing, et al. Analysis and optimization of the memory hierarchy for graph processing workloads[C]// Proc of the 25th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2019: 373–386

[40] Manocha A, Aragón J L, Martonosi M. GraphFire: Synergizing fetch, insertion, and replacement policies for graph analytics[J]. IEEE Transactions on Computers, 2022, 72(1): 291−304

[41] Vakil-Ghahani A, Mahdizadeh-Shahri S, Lotfi-Namin M-R, et al. Cache replacement policy based on expected hit count[J]. IEEE Computer Architecture Letters, 2017, 17(1): 64−67

[42] Vijaykumar N, Jain A, Majumdar D, et al. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory[C]// Proc of the 45th ACM/IEEE Annual Int Symp on Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 2018: 207–220

[43] Yao Pengcheng, Zheng Long, Zeng Zhen, et al. A locality-aware energy-efficient accelerator for graph mining applications[C]// Proc of the 53rd Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2020: 895–907

[44] Sardashti S, Seznec A, Wood D A. Skewed compressed caches[C]// Proc of the 47th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2014: 331–342

[45] Sardashti S, Seznec A, Wood D A. Yet another compressed cache: A low-cost yet effective compressed cache[J]. ACM Transactions on Architecture and Code Optimization, 2016, 13(3): 1−25

[46] Panda B, Seznec A. Dictionary sharing: An efficient cache compression scheme for compressed caches[C]// Proc of the 49th Annual IEEE/ACM Int Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society, 2016: 1–12

[47] Carvalho C D, Seznec A. Understanding cache compression[J]. ACM Transactions on Architecture and Code Optimization, 2021, 18(3): 1−27

[48] Liu Ning, Li Dongsheng, Zhang Yiming, et al. Large-scale graph processing systems: A survey[J]. Frontiers of Information Technology & Electronic Engineering, 2020, 21(3): 384−404

[49] McCune R R, Weninger T, Madey G. Thinking like a vertex: A survey of vertex-centric frameworks for distributed graph processing[J]. ACM Computing Surveys, 2015, 48(2): 1−39

[50] Nguyen D, Lenharth A, Pingali K. A lightweight infrastructure for graph analytics[C]// Proc of the 24th ACM Symp on Operating

Systems Principles. New York: ACM, 2013: 456–471

[51] Shun Julian, Blelloch G E. Ligra: A lightweight graph processing framework for shared memory [C] // Proc of the 18th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2013: 135–146

[52] Zhang Yu, Liao Xiaofei, Jin Hai, et al. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing [C] // Proc of the 26th USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2018: 441–452

[53] Zhang Yiming, Li Dongsheng, Liu Ling. Leveraging glocality for fast failure recovery in distributed RAM storage[J]. ACM Transactions on Storage, 2019, 15(1): 1−24

[54] Wang Kai, Zuo Zhiqiang, Thorpe J, et al. RStream: Marrying relational algebra with streaming for efficient graph mining on a single machine [C] // Proc of the 13th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2018: 763–782

[55] Wang Kai, Hussain A, Zuo Zhiqiang, et al. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code [C] // Proc of the 22nd Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2017: 389–404

[56] Maass S, Min C, Kashyap S, et al. Mosaic: Processing a trillion-edge graph on a single machine [C] // Proc of the 12th European Conf on Computer Systems. New York: ACM, 2017: 527–543

[57] Zhang Dunbo, Zeng Linlin, Wang Ruoxi, et al. Shuffle-SRAM: In-SRAM parallel bitwise data shuffle[J]. Journal of Computer Research and Development, 2025, 62(1): 75−89 (in Chinese)
(张敦博, 曾灵灵, 王若曦, 等. 混洗 SRAM: SRAM 中的并行按位数据混洗[J]. 计算机研究与发展, 2025, 62(1): 75−89)



Jia Chaoyang, born in 1995. PhD candidate. His main research interest includes computer architecture.

贾朝阳, 1995 年生。博士研究生。主要研究方向为计算机体系结构。



Chen Shi, born in 1999. PhD candidate. His main research interest includes computer architecture.

陈　湜, 1999 年生。博士研究生。主要研究方向为计算机体系结构。



Liu Jingyu, born in 1996. PhD candidate. His main research interest includes computer architecture.

刘京雨, 1996 年生。博士研究生。主要研究方向为计算机体系结构。



Shen Li, born in 1976. PhD, professor, PhD supervisor. His main research interests include multi/many-core architecture, processing-in-memory, and high performance computing.

沈　立, 1976 年生。博士, 教授, 博士生导师。主要研究方向为多核/众核体系结构、存内计算、高性能计算。