Improving Streaming Graph Processing Performance using Input Knowledge

Abanti Basak
abasak@ucsb.edu
UC Santa Barbara
USA

Zheng Qu
zhengqu@ucsb.edu
UC Santa Barbara
USA

Jilan Lin
jilan@ucsb.edu
UC Santa Barbara
USA

Alaa Alameldeen
alaa@cs.sfu.ca
Simon Fraser University
Canada

Zeshan Chishti
zeshan.a.chishti@intel.com
Intel Labs
USA

Yufei Ding
yufeiding@cs.ucsb.edu
UC Santa Barbara
USA

Yuan Xie
yuanxie@ucsb.edu
UC Santa Barbara
USA

ABSTRACT

Streaming graphs are ubiquitous in today's big data era. Prior work has improved the performance of streaming graph workloads without taking input characteristics into account. In this work, we demonstrate that input knowledge-driven software and hardware co-design is critical to optimize the performance of streaming graph processing. To improve graph update efficiency, we first characterize the performance trade-offs of input-oblivious batch reordering. Guided by our findings, we propose input-aware batch reordering to adaptively reorder input batches based on their degree distributions. To complement adaptive batch reordering, we propose updating graphs dynamically, based on their input characteristics, either in software (via update search coalescing) or in hardware (via acceleration support). To improve graph computation efficiency, we present input-aware work aggregation which adaptively modulates the computation granularity based on inter-batch locality characteristics. Evaluated across 260 workloads, our input-aware techniques provide on average 4.55× and 2.6× improvement in graph update performance for different input types (on top of eliminating the performance degradation from input-oblivious batch reordering). The graph compute performance is improved by 1.26× (up to 2.7×).

KEYWORDS

Graph analytics, Streaming graphs

ACM Reference Format:
Abanti Basak, Zheng Qu, Jilan Lin, Alaa Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. 2021. Improving Streaming Graph Processing Performance using Input Knowledge. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480096

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10.
https://doi.org/10.1145/3466752.3480096

1 INTRODUCTION

Streaming graph processing involves batched updates and analytics on graphs that evolve over time. This scenario is critical in applications such as graph convolutional networks [52], social network analysis [23, 37, 48], financial fraud detection [53], anomaly detection [30], and recommendation systems [28, 35, 61]. Streaming graph processing also lies at the heart of national security problems such as cyber attack detection and entity resolution. This has led to large-scale national initiatives (e.g., DARPA HIVE) to develop optimized streaming graph systems [1, 4, 5].

Effective handling of streaming graph data requires high performance solutions for 1) update (ingestion of new edges contained in input batches), and 2) compute (analytics on the latest snapshot of the graph). Numerous competitive streaming graph systems have recently proposed novel data structures and computation models [18, 20, 23, 26–28, 30, 32–35, 38–42, 44, 48, 53, 58–63, 67, 68]. The shortcoming of existing systems is that they do not consider the issue of input sensitivity which is critical to optimize both update and compute performances. Input batches may exhibit variations in structural properties, such as degree distributions of individual input batches or locality characteristics between consecutive input batches. These diverse input properties give rise to challenging trade-offs in software performance which, if ignored, can lead to a substantially sub-optimal performance. It is possible to significantly optimize the system performance through a design approach where input knowledge-driven adaptive software and hardware solutions complement each other. For example, batch reordering (RO) is a software optimization which reorganizes an input batch to remove lock-based operations in streaming graph updates [26, 42]. The state-of-the-art input-oblivious RO improves the update performance for *wiki*'s input batches by 2.7× but severely degrades the update performance for *uk*'s input batches (0.69×) (Fig. 1(a) and 1(b)). Our proposed input-aware adaptive software recovers *uk*'s lost update performance (from 0.69× to 0.92×) by capturing RO's

input-dependent performance trade-offs (Fig. 1(c)). Our proposed complementary input-aware hardware solution further increases *uk*'s update performance improvement to 1.60× (Fig. 1(d)), improving the overall system performance across both the workloads.

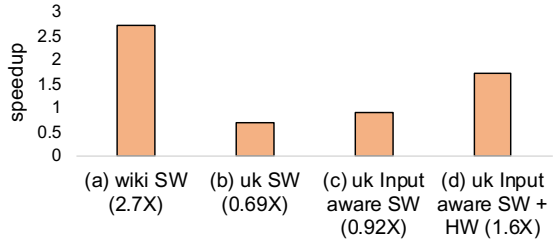


Figure 1: (a)/(b): Speedup in update performance from input-oblivious batch reordering for *wiki* and *uk* at input batch size=100K. (c)/(d): Input-aware software and hardware design recover and improve *uk*'s update performance. See Section 6.1 for benchmarks and evaluation setup.

For efficient graph updates, we propose input-aware batch reordering with software/hardware dynamic execution modes (Section 4). We characterize RO across 260 workloads and find that its impact on the update performance depends on the degree distribution of the input batches. Our experiments show that RO performance varies from high speedups to significant degradations. This study motivates the need to adaptively reorder incoming batches based on their input characteristics. We propose adaptive batch reordering (ABR) which uses a low-overhead online technique to collect information on the degree distribution of incoming batches. Specifically, we propose the concept of *order-λ clusterable average degree (CAD_λ)* which is used by ABR to predict whether an input batch is suitable for batch reordering. Compared to a naive always-RO solution, ABR can save the update performance from degradation in reordering-adverse cases without compromising the high speedup of reordering-friendly cases.

We propose two additional case-specific optimizations which complement ABR during the update phase: software-level update search coalescing (USC) for reordering-friendly cases and hardware-accelerated update (HAU) for reordering-adverse cases. USC leverages the degree distribution and the reordered organization of reordering-friendly input batches to substantially reduce the amount of search operations during edge updates. Since reordering clusters the incoming edges of a vertex, it is possible to search for all the incoming edges in the current edge data of the vertex *in one go*. ABR and USC cooperatively provide effective software optimizations for the updates of reordering-friendly input batches.

HAU complements ABR during the update phase of input batches with reordering-adverse degree distributions. Although ABR successfully recovers the RO performance degradation for these cases, it is unable to provide any additional benefit over the baseline. ABR turns off the optimizations of batch reordering and associated USC because their software overheads are expensive. Hence, reordering-adverse batches are still limited by 1) lock-based updates and 2) overheads of update search operations. HAU accelerates graph updates to remove these two bottlenecks. To remove lock-based software updates, HAU introduces enhancements to the cache controller and the on-chip processor-network interface to map each update task to a specific core. To mitigate search overheads, HAU uses simple dedicated logic in the cache controller to scan edge data cachelines, removing CPU instruction overhead for search operations. ABR and HAU cooperatively provide high-performance updates in reordering-adverse cases.

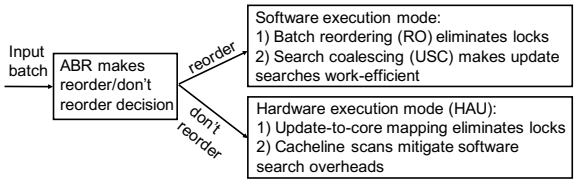


Figure 2: Input-aware SW/HW graph updates

As summarized in Fig. 2, ABR, USC, and HAU together optimize the update performance by dynamically adopting the best suited execution mode based on the input batch characteristics. Input batches with reordering-friendly degree distributions (identified by ABR) are updated in the software execution mode with two optimizations: batch reordering and USC. In contrast, reordering-adverse input batches are updated with HAU. We quantitatively show that input-aware dynamic software/hardware execution for graph updates outperforms input-oblivious updates with either exclusively software or exclusively hardware execution mode.

For higher streaming graph computation performance, we propose input-aware work aggregation (Section 5). Consecutive input batches sometimes exhibit a large overlap in graph modifications (i.e., inter-batch locality). Scheduling two separate computation rounds/phases to analyze these high-overlap input batches leads to work redundancy. We propose overlap-based compute aggregation (OCA) which adaptively modulates the computation granularity based on the inter-batch locality characteristics in input batches. OCA uses a low-cost online technique to identify inter-batch locality, and, for high locality, aggregates the computation. Thus, two input batches of similar graph modifications can be analyzed by scheduling a single computation round, increasing the compute efficiency.

2 NOVELTY AND IMPACT

We discuss the ways in which our overall approach and techniques advance the state-of-the-art prior work.

Streaming graph systems. We propose a novel perspective for efficient streaming graph processing: using input knowledge to optimize the performance for a given underlying data structure and computation model. This is orthogonal to the conventional approach [18, 20, 23, 26, 27, 27, 28, 30, 32–35, 38–42, 44, 48, 53, 58–63, 67, 68] of presenting a new system composed of input-oblivious data structures and/or computation models to outperform the state-of-the-art. We perform a novel characterization study across 260 workloads (Section 4.1) and show that input-oblivious software optimizations degrade performance in many cases. Our proposed input-dependent optimizations are applicable to most standard data structures and computation models.

Input-dependent graph processing. Existing input-aware solutions focus on static graph processing [7, 8, 11, 50] and are not readily reusable for streaming graphs. They 1) operate on *static input whole graph*, 2) typically make a decision once in the preprocessing

step, and 3) only capture the trade-offs for the graph computation phase. In contrast, the requirements for streaming graphs are substantially different. Our proposed ABR 1) operates on *input batches* whose size is much smaller than a whole graph, 2) makes online decisions multiple times during the execution on continuously incoming input batches, and 3) captures the performance trade-offs for the graph update phase (i.e., whether RO overhead pays off in terms of performance gains from lock-free updates). Hence, our designed ABR is an *extremely* lightweight (due to requirements 1 and 2) input-dependent algorithm with a novel and minimally intrusive CAD_λ metric which captures the update phase's performance trade-offs. Furthermore, another novelty in our input-dependent approach is using input batch characteristics to dynamically decide *between software and hardware* execution modes to achieve a higher update performance than a SW-only or a HW-only solution.

Hardware support for graph processing. Prior domain-specialized solutions for graph analytics are valuable but restricted to static graph computation [9, 10, 12, 24, 36, 45–47, 49, 51, 54, 57, 64, 65, 69–72]. Although a few papers [24, 45] provide APIs for graph updates, the discussion is limited to a subsidiary feature for a design targeted at static graphs and does not address the specific challenges for streaming updates. In contrast to prior proposals, we focus in-depth on the dynamic nature which is indispensable in real-world graphs. We tackle unique acceleration considerations which arise due to the characteristics in streaming graph updates. First, by treating input properties as the first-class design determinant, we provide insights that hardware acceleration is beneficial and meaningful for reordering-adverse input batches where software overheads are too high. Second, we selectively trigger HAU for these input batches to resolve the relevant challenges (overheads of software locks and search operations). Concerning HAU's specific design techniques (Section 4.4), HAU's concept of "task" bears some resemblance to GraphPulse's event-driven approach [54]. Although an important work for static graph computation, GraphPulse is not a drop-in replacement for HAU because 1) its events cannot represent and process streaming graph updates, 2) its design cannot solve update search overheads, and 3) it is a fully customized standalone ASIC, whereas HAU acceleration is CPU-coupled where the introduced changes are aware of the CPU architecture.

3 BACKGROUND

We describe the stages in the execution pipeline of streaming graph processing and the basics of batch reordering.

3.1 Streaming Graph Processing

The input to a streaming graph processing system is a stream of incoming edges. An input batch contains a given number of incoming edges represented by <*source*, *destination*> tuples (also *weight* for weighted graphs). Once a batch of edges enters the system, two stages are executed which provide newly computed results. First, the **update** phase ingests the incoming edges in the input batch into the graph data structure. Second, in the **compute** phase, an algorithm such as PageRank is performed on the latest snapshot of the graph data structure. The system handles dynamism by performing repeated and interleaved update and compute in response to continuous batches of incoming edges [23, 40, 44, 59, 63, 68].

The concepts of *vertex degree* and *degree distribution*, typically used to describe the whole graph structure of a static graph, can be extended to an input batch in a streaming graph. *Vertex degree* of v refers to the number of incoming edges of v in the input batch. The *degree distribution P(k) (or N(k))* of an input batch refers to the fraction (or number) of vertices in the input batch with degree k.

3.2 Batch Reordering (RO) Basics

RO is a pre-update operation where the input batch of edges is reorganized to ensure lock-free edge updates [26, 42]. In the baseline input batch, during parallel edge updates, two separate threads may update edges for the same vertex, requiring locks to protect against shared memory access conflict. In contrast, in the reordered input batch, the edges of the same vertex v are clustered (parallel stable sort from C++ Boost library [3]) and a carefully designed work division ensures that a specific thread updates all the input edges of v (dynamic OpenMP scheduling ensures load balancing).

The baseline (non-reordered) and RO-based update methodologies possess different benefits and costs, giving rise to RO's input-dependent performance trade-offs. *Baseline*: The benefit of the baseline is that it offers fine-grained edge-level parallelism and requires no change to the input batch format. The edge-centric work division is in perfect alignment with the input batch format. Incoming graph changes arrive as edges and the baseline treats the edge as the granularity of parallelism by assigning one thread per edge. However, the baseline's cost constitutes lock operations because separate threads may update edges for the same vertex. *RO*: RO's benefit involves completely eliminating locks by adopting vertex-centric updates (i.e., a thread updates all the edges of a given vertex). However, this comes at the cost of software overheads because the input batch format is inherently organized as edges instead of in a vertex-centric fashion. First, the input batch must be sorted to cluster edges belonging to the same vertex. Second, sorting should be carried out with respect to both source vertices and destination vertices to account for both in-edges and out-edges. This results in two reordered input batches which must each be updated separately. Finally, lock elimination involves additional scheduling overheads (scheduling must ensure that each thread updates all the edges belonging to a given vertex before moving on to another vertex in its task list). Our characterization in Section 4.1 shows how input batches of different degree distributions are affected by the relative costs and benefits of the two update methodologies.

4 INPUT-AWARE STREAMING GRAPH UPDATES

We first present our characterization study of the performance trade-offs of RO. After explaining ABR, USC, and HAU, we discuss input-aware SW/HW dynamic execution.

4.1 Characterization of RO Performance Trade-offs

Characterization across 260 workloads shows that the *performance from RO exhibits input sensitivity* (Fig. 3, left y-axis). *Topcats*, *talk*, *berkstan*, *yt*, *superuser*, and *wiki* indeed achieve up to about 3× improvement in update and overall performances at higher batch sizes of 100K and 500K (also at 10K for *talk*, *yt*, and *wiki*). However, at smaller batch sizes of 100 and 1K, these datasets experience

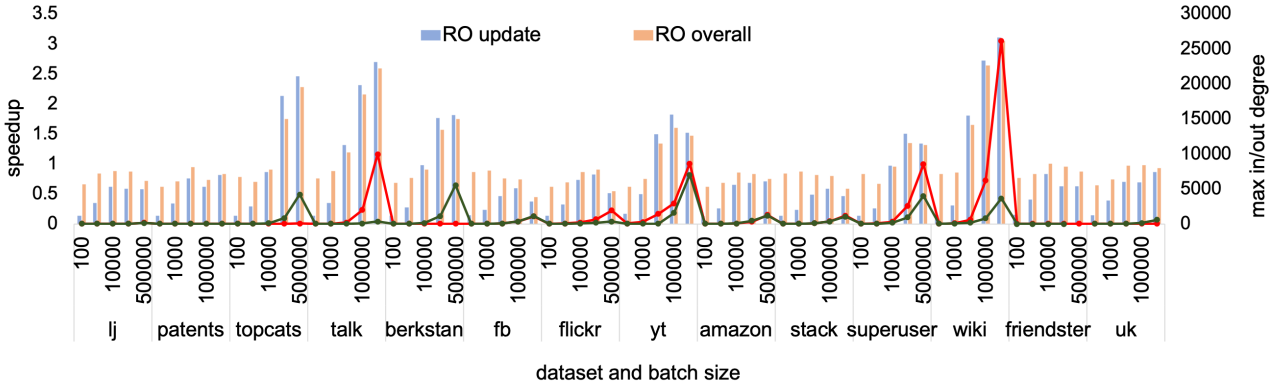


Figure 3: Left y-axis: Effect of RO on update and overall (i.e., update and compute combined) performances (see Section 6.1 for the evaluation methodology). Right y-axis: Maximum in/out degree in an input batch (average across all batches).

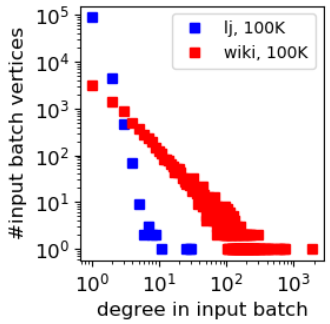


Figure 4: Input batch degree distributions of *lj* and *wiki* at batch size = 100K (log-log plot)

degradations in update and overall performances. The remaining datasets (*lj*, *patents*, *fb*, *flickr*, *amazon*, *stack*, *friendster*, and *uk*) experience performance degradation from RO at all batch sizes.

We observe that *high-degree input batches are reordering-friendly, whereas low-degree input batches are reordering-adverse.* We define "high (low)-degree input batch" as an input batch where the *top degrees* are high (low). For example, Fig. 4 shows the degree distributions of representative input batches of *lj* and *wiki* at batch size of 100K. *Lj*'s input batch is low-degree (e.g., top ten degrees lie in the range of 7-30, 30 being the maximum degree), whereas *wiki*'s input batch is high-degree (e.g., top ten degrees lie in the range of 401-1881, 1881 being the maximum degree)[1]. Since the maximum degree represents the upper bound of an input batch's top degrees, we use it as an indicator metric in Fig. 3 (right y-axis) to show the correlation with the RO performance (left y-axis). Compared to the reordering-adverse cases, the reordering-friendly cases exhibit a higher maximum in-degree or out-degree, indicating a high-degree input batch. For a given dataset, a smaller batch size naturally leads to a low-degree input batch (maximum possible degree = batch size). Therefore, small batches suffer from performance degradation when RO is applied (Fig. 3).

The performance trade-offs of RO can be understood by connecting the degree distribution of the input batches to the relative costs and benefits of RO (Section 3.2). High-degree input batches are reordering-friendly because:

- In the baseline, a large number of locks need to be acquired to update a top-degree vertex v. High-degree batch means v possesses a very high edge count. Updating each incoming edge of v needs a lock to be acquired on v's edge data because multiple threads may update v's incoming edges.
- In the baseline, in addition to the large number of locks described above, the cost of acquiring a lock is high for v. The cost of a lock acquisition involves waiting for another thread to finish updating an incoming edge for v. The wait time is proportional to the length of v's edge data array because updating involves a search scan for duplicate check (Section 4.3). The length of v's edge data array is large because v is a top-degree vertex in a high-degree input batch.

The above two factors together lead to high lock overheads for high-degree input batches in the baseline scheme. RO can eliminate these serious lock overheads. The cost of RO is small compared to the savings from baseline's lock overheads. In contrast, low-degree input batches are reordering-adverse because lock overheads in the baseline technique are not serious (i.e., top degrees are relatively small) and RO's extra software overheads are larger than the potential savings.

In addition to the above performance trade-offs, we find *temporal stability in the input batch degree distribution for a given dataset and batch size combination.* As shown with an example in Fig. 5 for *lj* at input batch size of 100K, the input batches consistently show stable degree distribution over time with increasing batch numbers. However, across different dataset-batch size combinations (*lj*-100K versus *wiki*-100K), the degree distribution is clearly different (Fig. 4). In the next section, we discuss how these insights are used to design a low-cost and effective adaptive batch reordering technique.

In addition to the above characterization, Fig. 6 shows that, on average (geomean), 19% and 33% of the total time is spent on graph updates in the baseline and RO, respectively. RO (input-oblivious

[1]**Terminology clarification:** We consistently use the term *top degree* to refer to an intra-input-batch large degree. In contrast, the term *high/low-degree* is used to differentiate between the top degrees across different input batches. Thus, a top-degree vertex in a high-degree input batch has a larger edge count than a top-degree vertex in a low-degree input batch.

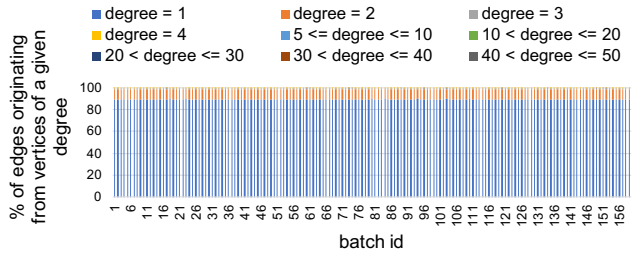


Figure 5: Input batch degree distribution over time for dataset LJ and input batch size 100K

batch reordering) increases the percentage of time spent on updates because many benchmarks are reordering-adverse (Fig. 3), motivating our proposed RO-targeted software and hardware optimizations (Sections 4.2-4.4). More recent streaming graph systems [26, 42] are equipped with system-specific techniques on top of RO (e.g., dual versioning, data compression, specialized memory allocation) which may reduce the update time to less than 33%. However, in this work, we isolate and study the RO technique.

4.2 Adaptive Batch Reordering (ABR)

ABR is an online technique that adaptively reorders input batches depending on their degree distributions. As shown in Fig. 7, ABR instruments the update phase of every n^{th} input batch (called ABR-active batch) to collect information on the input batch's degree distribution. Using this information, ABR makes a binary decision (reorder/don't reorder) and, for the next n update batches (called ABR-inert batch), the latest ABR decision is applied. ABR's overhead is low for two reasons:

- **Small number of ABR-active batches:** ABR leverages the temporal stability of input batch degree distribution (Section 4.1) to reduce required instrumentation. A reordering decision made by observing one ABR-active batch applies to many subsequent ABR-inert batches.
- **Low-cost degree distribution collection in ABR-active batches:** Instrumentation in ABR-active batches is overlapped with the actual edge updates. We propose a low-cost and minimally intrusive metric for instrumentation.

We propose a metric called ***order-λ clusterable average degree (CAD_λ)*** which is computed by ABR during instrumentation in the ABR-active batches to make accurate reordering decisions with small overhead:

$$order-\lambda \text{ clusterable average degree } (\text{CAD}_\lambda) = \frac{b-y}{x}$$

where,

b = input batch size

y = number of edges from vertices with $1 \leq \text{degree} \leq \lambda$

x = number of unique vertices with degree $> \lambda$

If $\text{CAD}_\lambda \geq TH$, reorder. Else don't reorder.

TH = some experimentally determined threshold.

ABR Algorithm: The following pseudocode represents the ABR algorithm:

```
|reordering = true      //default RO
|if (currentBatch is ABR-active) {
|    if (reordering == true) {
|        for each vertex v in batch {
|            count totalEdges for vertex v
|            if (1 <= degree(v) <= λ)
|                y = y + totalEdges
|            if (degree(v) > λ) x=x+1
|        }      // for
|    }      // reordering == true
|    else {      // reordering == false
|        for each edge in batch {
|                Populate concurrent hash map H with
|                (key: vertex ID; value: degree)
|            }
|        for each entry in hash map H {
|            if (1 <= degree(v) <= λ)
|                y = y + totalEdges
|            if degree(v) > λ
|                x=x+1
|        }
|    }      // reordering == false
|    Compute CAD_λ = (b − y)/x
|    if (CAD_λ >= threshold)
|        reordering = true
|    else reordering = false
|}     // ABR-active
|else {      // ABR-inert
|    if (reordering == true)
|        reorder incoming batch
|}     // ABR-inert
```

The concurrent hash map above is implemented using Intel TBB [6] so multiple threads may update edges for the same vertex. Above, x and y are incremented atomically.

CAD_λ intuition. CAD_λ is a measure of the *average degree of the top-degree vertices in an input batch* (i.e., the average degree computed using the intra-batch vertices with large edge counts). A high CAD_λ for an input batch indicates a high-degree input batch which is essential to achieve performance benefit from RO (Section 4.1). ABR decides to reorder if CAD_λ is greater than or equal to some experimentally determined threshold (TH). The design parameters of ABR are n (determines the instrumentation frequency), λ (distinguishes the top-degree vertices in the input batch), and TH (distinguishes between high CAD_λ and low CAD_λ). λ parameter is a cutoff applied to locate an individual input batch's top degrees. In contrast, TH parameter is used to understand the relative values of the top degrees across different input batches. Section 6.2.3 shows 1) how the parameter values are determined, and 2) the high decision-making accuracy.

Choice of CAD_λ and general applicability. CAD_λ fulfills the two essential requirements for a good metric: 1) high decision accuracy and 2) low overhead (see Section 6.2.3). We also considered other alternatives. For example, *average degree* exhibits poor decision-making accuracy. It is always a consistently low value because most vertices in an input batch possess low degrees. This obscures the distinction between high-degree/low-degree input batch. Rigorous mathematical measures of skewness and heavy-tailedness have been proposed in areas of network and probability theory

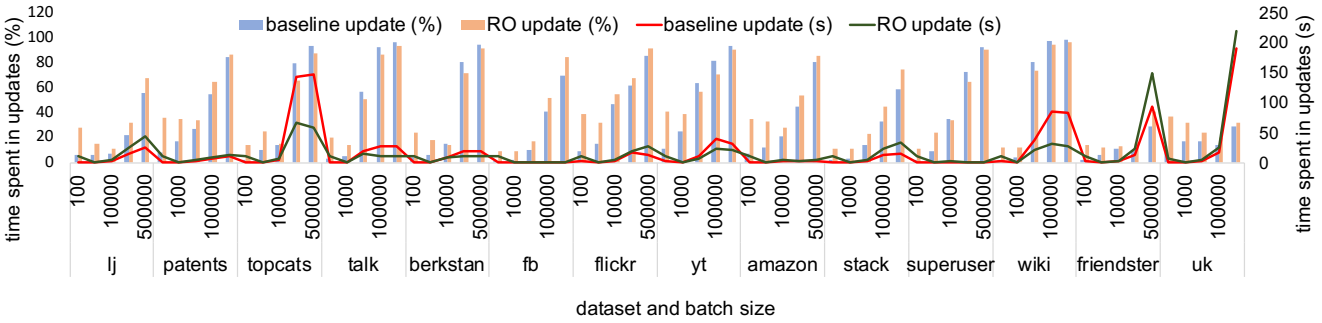


Figure 6: Total time spent in updates (percentage and absolute) for baseline and RO (always batch reordering) for all datasets and batch sizes (see Section 6.1 for the evaluation methodology)

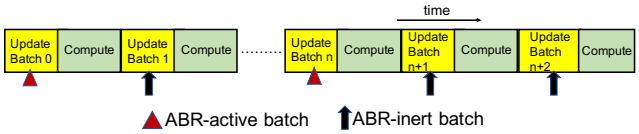


Figure 7: Adaptive Batch Reordering (ABR) design

[17, 19, 29]. However, they are computationally heavyweight for our streaming graphs scenario where measurements needs to be made online and multiple times. The lightweight and accurate CAD_λ is a better choice for practical system design where performance is a key metric. Moreover, these proposed measures [17, 19, 29] have been applied to large-scale whole graph instead of to input batches. A rigorous statistical analysis of their applicability to input batches is beyond the scope of this work. In addition to being accurate, practical, and low-overhead, CAD_λ is widely applicable. It has been developed by observing the examples of thousands of input batches from our large evaluation suite.

4.3 Update Search Coalescing (USC)

USC complements ABR in reordering-friendly cases to reduce update search overheads during duplicate checking. Duplicate checking is a common procedure in graph updates. Before updating an incoming edge $A \rightarrow B$, a search through the edge data of A checks for B so that B is not duplicated (the edge may have appeared in an earlier batch or may have already appeared earlier in the current batch). We identify that a high-degree input batch reordered by ABR provides the opportunity to substantially reduce the number of search scans for duplicates through search coalescing. Since a given thread updates all the input edges of a given vertex A, it is possible to search for *all of A's incoming target vertices during a single scan* of A's edge data. The effectiveness of USC depends on the underlying highly clusterable degree distribution (i.e., very high top degrees in high-degree batches). The higher the clusterability of the input batch, the higher the scope of search savings through search coalescing (see below and Section 6.2.3). These high-degree input batches are also reordering-friendly (hence reordered by ABR) and USC conveniently leverages their reordered data organization.

Fig. 8 shows the implementation of USC taking the example of updating three edges for source vertex A.

USC applies the following steps:

(1) As an update thread walks through the chunk of a reordered input batch consisting of the edges of A, it populates a small hash table with A's targets and weights (Section 6.2.3 shows this incurs little overhead).
(2) A's edge data are scanned only once. Each neighbor ID in the edge data array is searched for in the hash table using the neighbor ID as the key.
(3) A match in the hash table leads to updating the weight only (for weighted graphs). The specific target's entry is deleted from the hash table.
(4) Once the scan is complete, the remaining (non-matching) <*target*, *weight*> pairs in the hash table are inserted into A's edge array. Insertions are done either in some empty slot identified during the scan or at the end of the array.

In contrast, a non-reordered batch requires three separate scans through the edge data of A because multiple threads update the edges. Therefore, the higher the clusterability (i.e., per-vertex edges), the higher the benefits of USC. USC does not impact the granularity or amount of parallelism with respect to batch reordering. The contribution of USC is that it saves search-related work for individual threads participating in the parallel update process.

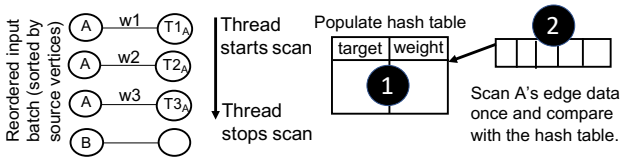


Figure 8: Overview of update search coalescing (USC)

4.4 Hardware-Accelerated Update (HAU)

HAU provides architectural support for the update of reordering-adverse input batches. Although their RO performance degradation is successfully recovered by ABR, they are still limited by 1) lock-based updates and 2) update search overheads. To resolve the former, HAU assigns each incoming update to a specific core. To resolve the latter, HAU uses specialized logic in the cache controller to scan cachelines, removing CPU instruction overhead for searches.

4.4.1 ***Design overview.*** (Fig. 9). The task-producing core triggers the HAU by feeding update tasks from the software ①. An

update task for an incoming edge *<src, target>* takes the form of *<src's edge data start address, src's current degree, target>*. It is routed through the network-on-chip (NOC) to a task-consuming core ②[2] obtained by *src* mod *N* where *N* is the number of task-consuming cores (Section 4.4.3 discusses this algorithm). The task-consuming core's cache controller uses the task description received from the NOC to fetch the edge data cachelines ③. Upon each cacheline's return to the L1D cache, the cache controller captures and searches it for *target* (duplicate check) using its dedicated scan logic ④. For example, we consider a toy graph consisting of vertices *V0*, *V1*, *V2*, and *V3*. We also consider a small input batch of size 1 consisting of the incoming edge *V1->V2*. The update task *<V1's edge data start address, V1's current degree, V2>* is assigned to core 1 (*V1* mod 2). Core 1's cache controller fetches *V1*'s edge data cachelines, searches for the target *V2* before updating the edge. A similar set of operations needs to be performed to update *V2*'s edge list to include *V1*.

4.4.2 ***Design details.*** We take the reference network interface in [25] and highlight our enhancements (Fig. 10/11).

Task production (Fig. 10): Driven by the software, the core initiates a request of a new type called *task* ① (already existing types are *read*, *write*, etc.). The address corresponding to this request is the start address of the edge data of vertex *src* and the data fields encode *src*'s current degree and *target*. We use the *tag* field to encode the destination core ID (in a conventional request, this field helps the core identify the corresponding reply). Unlike a *read* or *write* request, the address field of a *task* request does not mean this core expects data from this address. It only encodes some information, together with the data field, that needs to be sent to another core. The control flow for a *task* request involves bypassing caches ② and initializing a new miss status handling register (MSHR) entry with a new type of status called *task pending* ③. Allocating an MSHR entry is essential because the *message transmit unit* only reacts to MSHR status changes [25]. The *message transmit unit* formats the NOC message of the update task and injects it into the network ④. The MSHR entry status is changed to *idle* and it is freed ⑤.

Task consumption (Fig. 11): Upon receiving a *TaskReq* message from the NOC at the message receive unit ①, a new MSHR entry is allocated with the status *task received* ②. The protocol FSM takes appropriate actions on the MSHR status change: the task is forwarded to a FIFO buffer to the cache controller ③ and the MSHR entry is freed ④. Simple dedicated logic in the controller ⑤ uses the *edge data start address* to fetch the edge data cachelines. Each returning cacheline to the L1D cache is scanned to check for the *target* node. If found, the loop stops. Otherwise, the controller brings in consecutive cachelines until the number of scanned elements matches the *degree*. If the *target* is not found even after the entire edge data has been exhausted, the controller hands over the write operation to the core through the FIFO buffer ⑥. The core takes over this action because new memory region may need to be allocated to accommodate the *target*.

4.4.3 ***Discussion.*** We discuss HAU's important details.

Task assignment: Hashing-based task scheduling is very low-cost and requires no tracking overhead of scheduling history or progress status of large core counts in a modern multicore architecture. Moreover, this scheduling ensures that all incoming edges for vertex v are updated at the same core where v's edge data resides, implicitly guaranteeing safety against race conditions from concurrent accesses (allowing us to eliminate software lock overheads). A more complex assignment requires expensive design to explicitly guarantee race-safe accesses (e.g., GraphPulse [54] requires additional cycles to coalesce events for identical vertex in large hierarchical queues). Minimizing design complexity and scheduling overheads is critical because, in contrast to static whole graph processing [54], acceleration granularity for dynamic graph updates is a much smaller input batch, making the design constraints tighter. Section 6.2.3 discusses workload distribution.

Interaction with SW: To achieve the interaction between software and HAU, we adopt the technique used in previous work [46] where low-level software API methods translate to two specific instructions. On the task-sending side, the core uses a `supply_task` instruction to communicate the information related to the update task to HAU. On the task-receiving side, the core uses a `fetch_task` instruction to get the information from the FIFO buffer.

Virtual memory: The cache controller logic requires virtual-to-physical address translation for the address it obtains from the task description in the FIFO buffer. We adopt the approach of previous work [10, 46] to ensure this. The cache controller logic shares the core's address translation machinery and handles page faults like [46].

Coherence protocol: HAU does not affect the cache coherence protocols. An update *task* request is a cache-bypassing point-to-point push-style communication between well-defined sender and destination cores and does not involve any additional coherence messages (Fig. 10/11). To process the *task*, changes are confined to the cache controller after edge data cachelines return through traditional coherence protocols.

Update ordering: HAU maintains the same consistency as software graph updates because, in the execution model we consider, the programmer expects that consistency is guaranteed at the granularity of an input batch. In a batch, individual updates (i.e., incoming edges) for vertex v may arrive and be processed at task-consuming core c in any order. The final result (i.e., at the end of the update phase for this input batch) is the same (and equivalent to software updates) because all the updates show up in v's edge data (the order of showing up does not matter). To maintain consistency in case the input batch contains both edge insertions and deletions, software triggers HAU to perform all insertions first before performing deletions (deletion requires that the edge already exists). Since tasks are independent, this update ordering policy ensures that updates are deadlock-free, i.e., no circular dependencies exist between any subset of tasks.

MSHR management: Keeping the baseline NOC topology, routing, and buffering unchanged, we introduce a new request/response type and show how it fits into the reference processor-network interface [25]. We add ten new MSHR entries (2× increase) reserved for outgoing/incoming *tasks* to avoid MSHRs becoming a performance bottleneck. *Task*-MSHRs are proactively freed, making space for new tasks. A *task pending* MSHR is freed as soon as the *task* is released into the network (Fig. 10 ④, ⑤). A *task received* MSHR is

[2] 1) A core can be both task-producing and task-consuming. Fig. 9 illustrates a decoupled behavior only for clarity. 2) For weighted graphs, HAU includes an extra field *weight* in the update task.

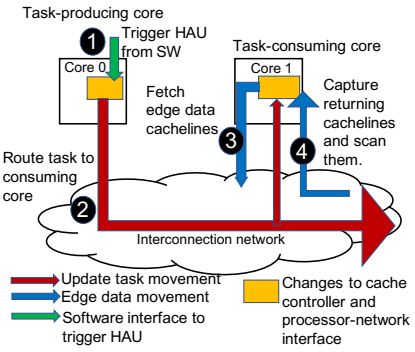


Figure 9: HAU overview

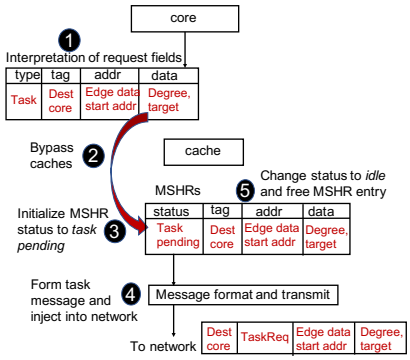


Figure 10: Task production

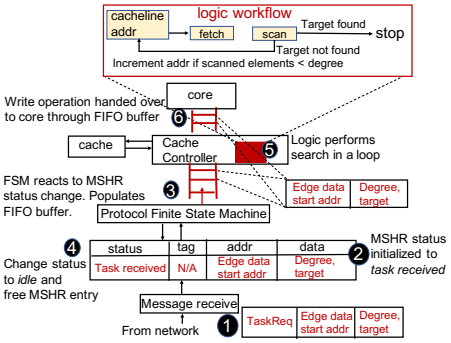


Figure 11: Task consumption

freed as soon as the FIFO buffer is populated (Fig. 11 ③, ④). The total volume of *task* traffic is limited to input batch size, which is much smaller than a whole graph (Fig. 3).

Hardware overhead: HAU incurs small hardware overhead. Using McPAT integrated with Sniper [21], the area of the baseline chip (Table 1) is 212 mm^2 in a 22nm technology node. We implement the cache controller logic in RTL and synthesize it with Synopsys Design Compiler. We obtain an area of 0.0058 mm^2, leading to an overhead of ~0.044%. Each FIFO buffer entry consists of four 64-bit fields (fourth field is *weight* to account for weighted graphs). Ten new MSHR entries and two 32-entry FIFO buffers lead to an additional 1KB and 2KB storage per core tile, respectively.

Generality: Since graph processing is an important application domain, domain specialization for higher performance and efficiency is a common approach in previous work on static graphs [10, 12, 31, 45, 46, 56]. HAU follows the similar approach of specialization to handle the *more general* case of dynamic graphs which is important but remains unexplored in prior proposals.

4.5 Input-Aware SW/HW Dynamic Execution

To address the software performance trade-offs arising from input sensitivity (Section 4.1), we adopt input-aware SW/HW dynamic execution for optimized performance and efficiency across all input types (experiments in Section 6.2.2). A SW-only approach (i.e., RO+USC) is sub-optimal for low-degree input batches because the SW overheads are higher than the performance gains. Without RO and USC, low-degree batches are still bottlenecked by software locks and search. HAU removes these bottlenecks and further improves graph update performance for low-degree batches. A HW-only approach (HAU) is sub-optimal for high-degree batches because HAU design is sophisticated only enough for low-degree batches, minimizing its hardware overhead and functional complexity. For example, HAU does not support search coalescing because it is not necessary (e.g., *lj's* input batches mostly contain of degree=1 vertices (Fig. 5), making search coalescing superfluous and a source of inefficiency). Adding more functionality to HAU is possible but only increases engineering effort, design complexity, and overhead when effective software solutions are realizable.

5 INPUT-AWARE STREAMING GRAPH COMPUTATION

Input-aware computation aggregation adaptively modulates the streaming computation granularity during the runtime based on the locality characteristics between consecutive input batches. Fig. 12 explains how it differs from the baseline computation workflow. In the latter (Fig. 12 (a)), update and streaming computation are interleaved like numerous previous streaming graph systems [23, 40, 44, 48, 59, 63, 68]. Once a batch of updates are applied to the graph, an algorithm re-executes on the latest snapshot of the graph to reflect the changed data structure. Thus, the update batch size indicates the streaming computation granularity because the computation considers the changes to the graph data structure caused directly and indirectly by an amount of modifications equal to the batch size. On the other hand, the proposed input-aware computation aggregation (Fig. 12 (b)) uses overlap-based compute aggregation (OCA) technique to adaptively increase the computation granularity when there is high inter-batch locality, i.e., high overlap between the graph modifications contained in batches n and $n+1$. OCA increases compute efficiency for high inter-batch locality because TC_{agg} is less than $TC_{\text{n}} + TC_{\text{n+1}}$, i.e., aggregating computation helps amortize the scheduling and data access overheads of launching two separate computation rounds.

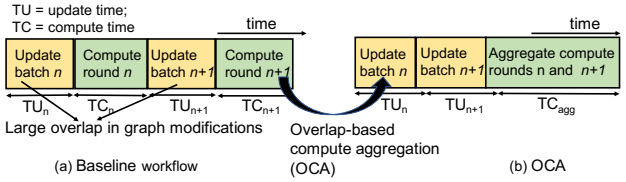


Figure 12: Overview of OCA

Design details. We classify the inter-batch locality between batches n and $n+1$ to be high when a large percentage of the unique vertices in batch $n+1$ also appeared in batch n (i.e., edge updates affect a lot of identical vertices across the two batches). This is reasonable because incremental computation models concentrate computation at or around the affected vertices. So, identical affected vertices across two input batches means that consecutive computation rounds touch similar regions of the graph. Scheduling two separate computation rounds to perform operations on similar regions of the graph leads to work redundancy in scheduling and

data accesses. Computation aggregation eliminates this redundancy with a single aggregated round. We implement a low-cost online mechanism for measuring inter-batch locality. The graph representation is augmented with an additional per-vertex field `latest_bid` which tracks the last batch where a vertex appeared. This field is updated along with edge updates during each update phase. During an ABR-active batch (Section 4.2) (batch $n + 1$), an update for vertex *src* increments a global counter `overlap_counter` if the `latest_bid` field for *src* reads n. In addition, another global counter `node_counter` is incremented to record the total number of unique vertices that appear in the ABR-active batch. When the updates of the ABR-active batch are over, the ratio of `overlap_counter` and `node_counter` provides the value of inter-batch locality. It is high if it exceeds a certain threshold which is determined empirically as follows: starting from a high threshold of 0.5, we progressively decrease the threshold and note the batch sizes where aggregation is activated and the corresponding level of speedup. We choose a value of 0.25 where most of the larger batch sizes experience high performance improvement. Below 0.25, we note that aggregation is triggered for smaller batch sizes. However, we avoid granularity aggregation for smaller batch sizes (see below). In addition, the speedup from these smaller batch sizes is small due to a low overlap. For example, *yt* dataset at a batch size of 10K experiences computation aggregation at a threshold of 0.15 (but not at higher thresholds) but the corresponding speedup is only 8%.

Application scenarios. Extremely latency-sensitive applications (e.g., security applications such as financial fraud detection) utilize a fine-grained computation granularity or small batch size for faster reaction to graph modifications. Trading off granularity for a higher computation performance is not a good choice in these application scenarios. We experimentally show that the adaptive OCA deactivates at small batch sizes and only activates at relatively larger batch sizes. A larger batch size indicates an application scenario which can trade off some granularity for a higher compute efficiency. Moreover, when OCA is activated, we coarsen the granularity by only one additional batch size worth of graph modifications. In addition, OCA is an adaptive optimization and can be easily entirely turned off if the application does not tolerate any sacrifice in granularity even for the larger batch sizes.

6 EVALUATION

6.1 Experimental Setup and Methodology

ABR, USC, and OCA are evaluated on a dual-socket Intel Xeon Platinum 8180 (Skylake) server with a total of 112 hardware execution threads (28 cores per socket, 2-way SMT). The server contains 38.5MB last-level cache per socket and 768GB memory. Performance evaluation of HAU is done on Sniper-7.2 [21] with the baseline architecture in Table 1.

Table 2 shows the evaluated datasets. The first seven (*talk-uk*) are static datasets that are randomly shuffled to break any ordering in the input files (they are often ordered in increasing source vertex ID, which is not the likely scenario of edge appearance for real-world streaming graphs). The remaining datasets (*fb-wiki*) are timestamped, i.e., the input file specifies the order in which the edges appear in the graph. We use SAGA-Bench [13] and perform experiments on the adjacency list data structure because it is used in multiple existing systems [32, 42, 67]. Four algorithms are evaluated: incremental PageRank (PR), incremental Single Source Shortest Paths (SSSP), static PR (start-from-scratch), and static SSSP. SAGA-Bench uses the computation model proposed in prior work [23, 67] for incremental algorithms and takes the static versions from GAP [14]. The evaluated input batch sizes are 100, 1K, 10K, 100K, and 500K. Combining 14 datasets, 5 batch sizes, and 4 algorithms, we run 260 workloads. The largest datasets *friendster* and *uk* are run on only the incremental algorithms because prior work [13] has shown that incremental compute models provide significantly better performance for larger datasets. The speedup in update/compute performance for each workload represents the ratio (between the baseline and the proposed technique) of the total update/compute time across all the batches (we start from an empty graph). Real hardware experiments of ABR, USC, and OCA are repeated three times. For simulation-based evaluation of HAU, it is not feasible to perform experiments as extensively as on a real hardware. Each of the 260 workloads consists of hundreds of batches; months of simulation time would be required. Therefore, we evaluate HAU on a subset of 8 datasets and 4 batch sizes (Table 3). The datasets cover different sizes (vertex/edge counts) and types (shuffled/timestamped).

Table 1: Simulated Baseline Architecture on Sniper-7.2

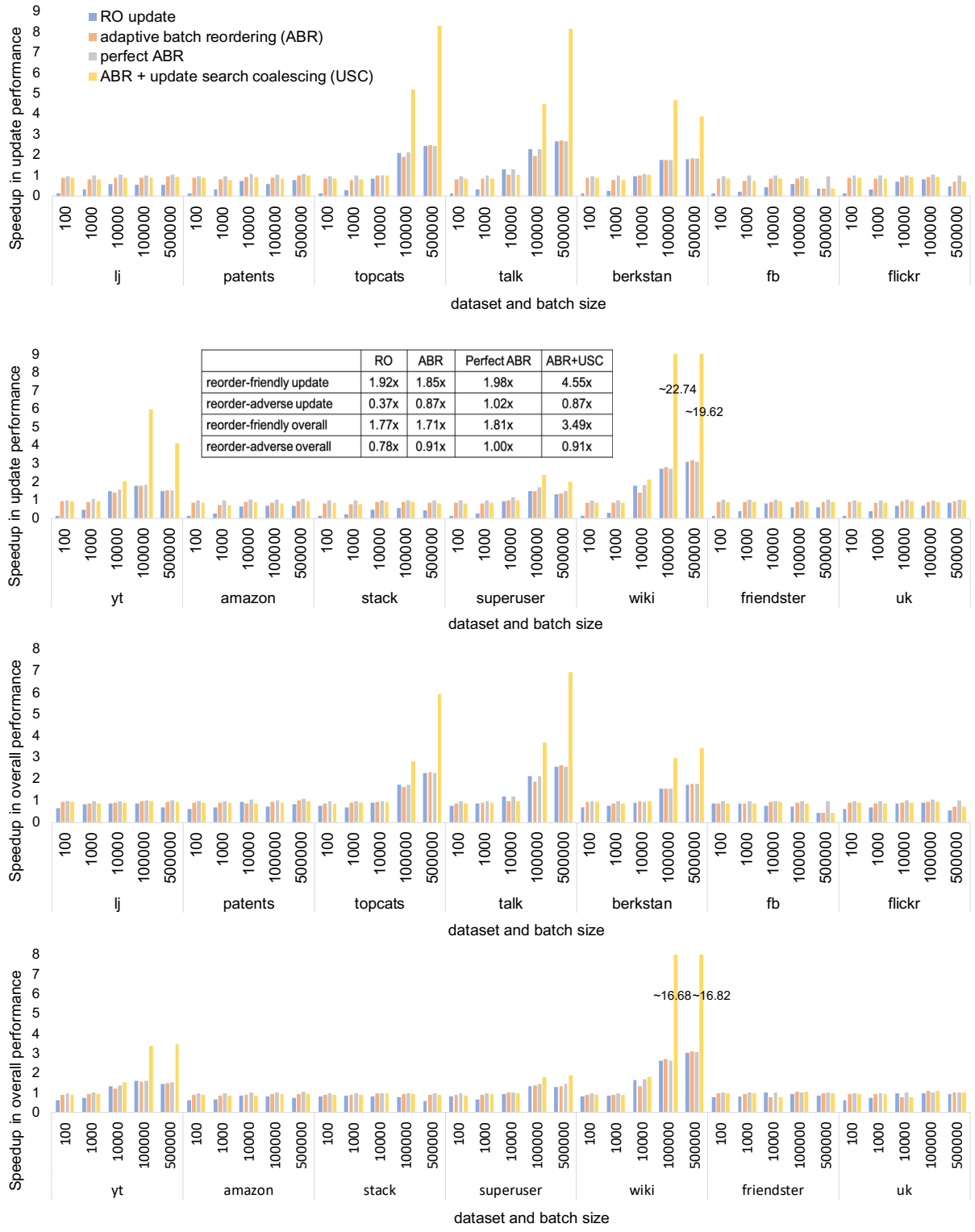
core	16 cores, 2.5GHz, 4-issue
L1D/I	32KB private, 8-way, 3 cycles
L2	256KB private, 8-way, 8 cycles
L3	16MB NUCA (2MB slices), 16-way, 8 cycles bank access latency
NOC	4x4 mesh, 2-cycle hop, per-link per-direction bandwidth = 256 bits/cycle
DRAM	4 memory controllers, 17GB/s per controller, 40ns device access latency, queue delay modeled

Table 2: Evaluated Datasets

dataset (short name)	vertices	edges
Wiki-Talk (Talk) [43]	2,394,385	5,021,410
WebBerkStan (BerkStan) [43]	685,230	7,600,595
cit-Patents (Patents) [43]	3,774,768	16,518,948
Wiki-Topcats (Topcats) [43]	1,791,489	28,511,807
soc-LiveJournal (LJ) [43]	4,847,571	68,993,773
com-Friendster (Friendster) [43]	65,608,366	1,806,067,135
UK-Union-2006-2007 (UK) [2, 16]	133,633,040	5,507,679,822
Facebook-wall (FB) [66]	46,952	876,993
Flickr-photo (Flickr) [22]	11,730,773	34,734,221
Youtube (YT) [55]	3,223,589	12,223,774
Amazon-ratings (Amazon) [55]	2,146,057	5,838,041
Stack-overflow (Stack) [43]	2,601,977	63,497,050
Superuser (Superuser) [43]	194,085	1,443,339
Wiki-talk-temporal (Wiki) [43]	1,140,149	7,833,140

6.2 Experimental Results

6.2.1 ***Performance.*** Fig. 13 shows that ABR does not substantially compromise the high RO update performance of reordering-friendly cases. As summarized in the table-inset, the geometric mean across reordering-friendly cases shows that the update



	RO	ABR	Perfect ABR	ABR+USC
reorder-friendly update	1.92x	1.85x	1.98x	4.55x
reorder-adverse update	0.37x	0.87x	1.02x	0.87x
reorder-friendly overall	1.77x	1.71x	1.81x	3.49x
reorder-adverse overall	0.78x	0.91x	1.00x	0.91x

Figure 13: Speedup in update (top two charts) and overall (bottom two charts) performance from ABR and USC. Each bar is the average across runs with different algorithms. ABR parameters are n=10, λ=256, and TH=465 (Section 6.2.3). The inset-table shows the average update/overall performances for both the categories (averaged across all dataset and batch size combinations which fall under the given category).

Table 3: Speedup in update, overall (average), and overall (max) performances from ABR+USC+HAU (normalized to ABR+USC).

Dataset	lj				patents				topcats				berkstan			
Batch Size	100	1K	10K	100K	100	1K	10K	100K	100	1K	10K	100K	100	1K	10K	100K
Update	3.32	3.99	3.17	1.84	2.73	4.09	2.11	3.44	1.14	2.16	1.45	1	1.48	2.46	1.82	1
Overall (average)	1.02	1.04	1.03	1.03	1.02	1.08	0.96	1.11	1.00	1.06	1.03	1	1.01	1.04	1.09	1
Overall (max)	1.04	1.09	1.06	1.07	1.05	1.15	1.10	1.22	1.03	1.30	1.08	1	1.03	1.18	1.14	1
Dataset	fb				flickr				amazon				superuser			
Batch Size	100	1K	10K	100K	100	1K	10K	100K	100	1K	10K	100K	100	1K	10K	100K
Update	1.88	3.22	1.88	2.90	2.87	7.54	4.47	1.96	2.45	4.59	2.27	2.10	1.44	2.94	1.69	1
Overall (average)	1.01	1.03	1.05	1.23	1.06	1.48	2.01	1.68	1.02	1.12	1.06	1.11	1.00	1.03	1.06	1
Overall (max)	1.02	1.04	1.21	1.30	1.22	1.77	3.21	3.29	1.07	1.21	1.11	1.29	1.01	1.07	1.10	1

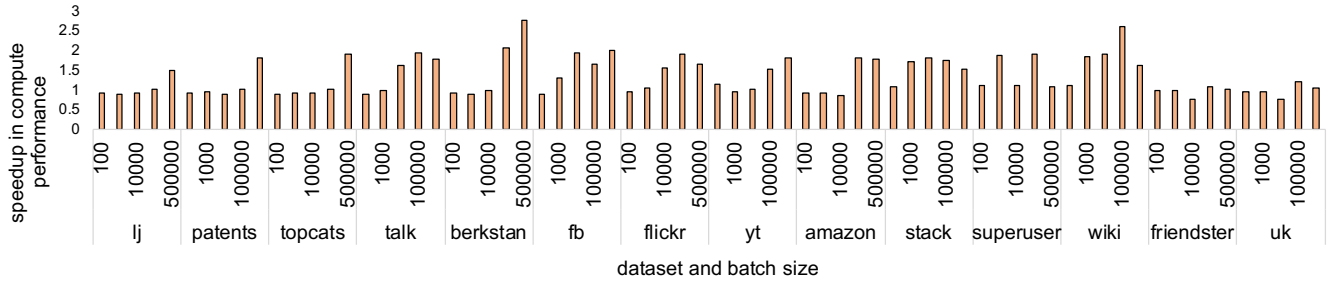


Figure 14: Speedup in compute performance from OCA for all datasets and batch sizes

speedups of always-RO and ABR are 1.92× and 1.85×, respectively. For reordering-adverse cases, ABR successfully recovers the update performance from degradation in a naive always-RO solution. Geometric mean across reordering-adverse cases shows that ABR pushes up the update performance closer to the baseline (0.37× to 0.87×). The table-inset, together with the bottom two charts, shows that the benefits of ABR are carried over to the overall performance (i.e., update and compute combined), providing evidence that graph updates have an important contribution to the overall performance (Fig. 6). ABR saves the overall performance from degradation for reordering-adverse cases (0.78× to 0.91×) while minimally disturbing the overall speedup of reordering-friendly cases (1.77× to 1.71×). Fig. 6 shows that, for reordering-adverse cases, RO increases the percentage of execution time spent in update. ABR helps such reordering-adverse cases and recovers most of the performance loss. The *perfect ABR* bars and inset column show that ABR performs close to a perfect adaptive technique with zero overheads. ABR performs at 93%, 85%, 94%, and 91% of perfect ABR for reordering-friendly update, reordering-adverse update, reordering-friendly overall, and reordering-adverse overall performance, respectively. For the updates of reordering-friendly input batches, ABR and USC together provide average speedups of 4.55× (max 23× for *wiki*-100K and 20× for *wiki*-500K) and 3.49× (max 17× for *wiki*-100K, 500K) in update and overall performance, respectively. Reordering and USC software optimizations are not applied on reordering-adverse cases. Instead, HAU complements ABR in these scenarios to improve the update performance (Table 3). The update speedup obtained from ABR+USC+HAU is normalized to ABR+USC running on the simulated architecture in Table 1 (note that ABR+USC+HAU means reordering-adverse input batches undergo ABR and HAU, whereas reordering-friendly ones undergo ABR and USC). Compared to ABR only, HAU provides on average 2.6× (max 7.5×) improvement in update performance across the reordering-adverse cases. For each case, the average and maximum improvements in overall performance are also shown in Table 3. The overall performance improvement from HAU depends on the importance of the update phase in the overall latency (Fig. 6 shows the percentage breakdown). For cases like *patents*-100K, *fb*-100K, *flickr*-10K, *flickr*-100K, and *amazon*-100K, the update phase is costly due to the large batch size, leading to a relatively higher overall performance improvement (e.g., 11% average and 29% max for *amazon*-100K). Usually, for smaller batch sizes, the update operation is relatively less costly because it is limited to a few input edges, explaining the relatively lower overall performance improvements. However, several smaller batch sizes do experience high overall performance improvements (e.g., *flickr* and *amazon*). Note that Table 3 does not include the batch size of 500K (reason explained in Section 6.1) where the update phase is generally very costly (Fig. 6). Larger overall performance improvements are expected at 500K batch size. HAU is not applied to reordering-friendly *topcats*-100K, *berkstan*-100K, and *superuser*-100K, as shown by the 1× speedup. They are executed in software mode with reordering and USC optimizations (Fig. 13).

For streaming graph computation, OCA is activated in cases of higher inter-batch overlap and can provide up to 2.7× speedup in compute performance (Fig. 14). Averaged across all datasets and batch sizes, the performance benefits experienced by incremental PR and incremental SSSP are 1.24× and 1.26×, respectively. OCA is predominantly triggered at relatively larger batch sizes (a desirable feature as explained in Section 5). This happens because of: 1) inherent traits of large batches, and 2) our choice of a relatively high overlap threshold. Larger input batches inherently contain larger number of vertices, leading to an increased possibility of high overlap in unique vertices between consecutive batches. Smaller input batches also exhibit some inter-batch overlap which fails to satisfy the overlap threshold.

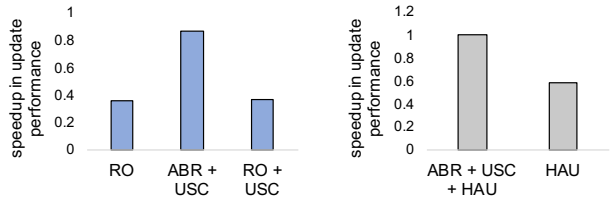


Figure 15: Left: Extension of Fig. 13 across reordering-adverse cases; shows impact of enforcing software optimizations (RO+USC). Right: Extension of Table 3 across reordering-friendly cases; shows impact of enforcing HAU.

6.2.2 ***Dynamic SW/HW graph updates.*** We quantitatively show that input-aware SW/HW execution mode outperforms an input-oblivious HW-only or SW-only update technique (see Section 4.5 for insights and explanation). SW-only: Our input-aware solution deviates from an input-oblivious SW-only solution by applying HAU on reordering-adverse input batches. We instead enforce RO+USC on them (Fig. 15 (left)) and find that RO+USC performs almost as poorly as RO. Since ABR+USC outperforms RO+USC (Fig. 15 (left)) and ABR+USC+HAU outperforms ABR+USC (Table 3), it follows that ABR+USC+HAU outperforms RO+USC. HW-only: Our solution deviates from a HW-only solution by dynamically applying RO+USC on reordering-friendly input batches. We extend Table 3 by enforcing HAU on them and show that the performance degrades (Fig. 15 (right)).

6.2.3 ***Further analysis.*** We further quantitatively analyze different techniques. Any overheads analyzed in this section are already included in the speedups reported in Section 6.2.1.

ABR and OCA overheads (Fig. 16): Reordered ABR-active batches experience negligible overhead (0.90×) due to CAD_λ collection. Non-reordered ABR-active batches experience a higher overhead on average (0.54×) because of instrumentation with a concurrent hash map. However, a small number of ABR-active batches ensures a small combined overhead across all batches. Normalized to ABR+USC, the average overhead incurred by OCA is very small (Fig. 16 (b)).

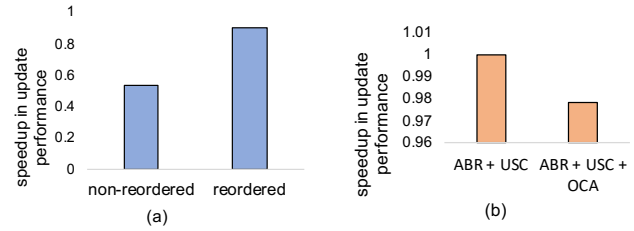


Figure 16: Overhead of (a) ABR and (b) OCA

USC insights and overheads: We study the examples of *superuser*-100K and *wiki*-500K to provide two key insights:

- High/low-degree input batches: *Wiki*-500K predominantly achieves a larger speedup than *superuser*-100K (Fig. 17) because the input batches of the former are high-degree in terms of both CAD_λ (1072 vs. 528) and maximum degree (43992 vs 3171). The exception are the first two batches of *wiki*-500K which are low-degree and where the graph is small (see below). A high-degree input batch means more coalescing, leading to more search savings.
- Negligible overhead: USC does not degrade the update performance even when the scope of speedup is smaller. This provides evidence that USC incurs negligible overhead of preparing the hash table (Fig. 8).

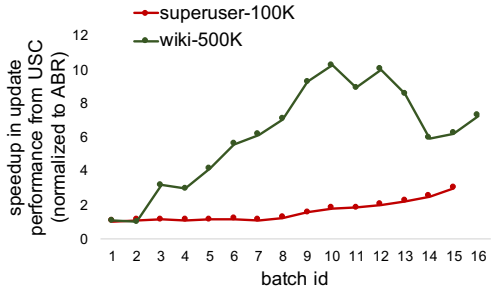


Figure 17: Temporal speedup from USC

ABR parameters/accuracy: The design parameters of ABR are n, λ, and TH. By analyzing the batches from different combinations of dataset and batch size, we choose the combination of λ and TH which maximizes the decision-making accuracy at 97% (λ=256 and TH=465) (Fig. 18(a)). The parameter n impacts both the decision accuracy and the overhead. A large n can reduce ABR overhead by reducing the frequency of instrumentation. However, it leads to coarse-grained decision-making which can miss temporal fluctuations in degree distributions, compromising ABR accuracy and performance. Fig. 18(b) shows that a larger n leads to a slightly better update performance *on average* (1.04× at n=10 versus 1.07× at n=100). However, *flickr*-500K, *yt*-100K, and *stack*-500K experience a poorer performance. For example, *stack*-500K has 127 batches in total, leading to 2 ABR decisions at n=100 and 12 ABR decisions at n=10. Therefore, n=100 misses some over-time fluctuations.

The ABR parameters are found by expanding the search space until a good enough ABR accuracy is attained. Fig. 18 (a) leaves out three datasets: *yt*, *friendster*, and *uk* since ABR is effective at recommending reordering decisions for these three datasets at all batch sizes (Fig. 13). To ensure parameter robustness, we choose a large number of workloads (after excluding *yt*, *friendster*, *uk*) when determining ABR parameters (11 datasets each at 5 batch sizes resulting in hundreds of example batches). Since these datasets are widely used in the graph research community and are representative of real-world scenarios, we believe that our developed parameters should be sufficiently robust. In future work, ABR could be extended with an online feedback tuning method.

HAU analysis: We study *uk*-100K at a batch number of 100. For work distribution, the maximum value of vertices/core (core 6) is 3% higher than the minimum (core 11) and 1.3% higher than the average across all cores (Fig. 19; since core 0 hosts the master thread in SAGA-Bench setup, we show the information on cores 1-15 which host the worker threads for graph updates). The maximum number of cachelines accessed per dedicated cache controller logic (core 13) is 600% higher than the minimum (core 11) and 148% higher than the average across all cores. A heavy-workload core does not straggle substantially because HAU eliminates i) remote cache accesses and ii) CPU instruction overheads for searches. In other words, HAU substantially minimizes time per unit of work

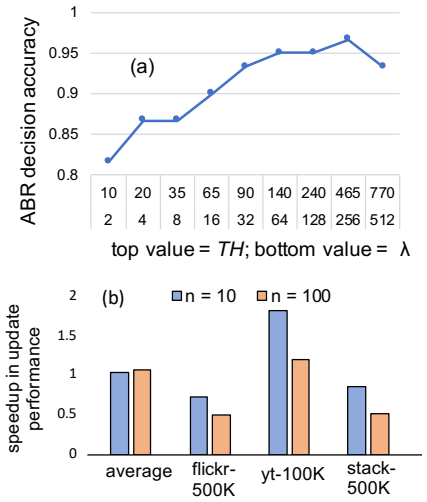


Figure 18: (a) ABR accuracy as a function of λ-TH combination. (b) Sensitivity of update performance to n.

(cacheline access + cacheline search), ensuring that non-uniform work distribution is not the most significant performance limiter for HAU (Table 3 shows HAU's existing design can achieve on average 2.6× update performance improvement). First, our update task assignment ensures that 98%-99% of the accessed edge data cachelines hit in the local core tile (Fig. 20), eliminating straggling due to more expensive remote cache accesses. In fact, HAU eliminates all remote cache accesses that would otherwise be present in the baseline software updates. Second, specialized logic in the cache controller eliminates the overheads of several CPU instructions for searches, limiting straggling due to time-consuming searches. We believe that, in future, HAU can easily be optimized with well-known load balancing schemes such as work-stealing [15]. Finally, Fig. 20 shows the impact of using NOC for update task distribution. The increase in average packet latency is within 10%, and some cores also experience a decrease in packet delay, depending on the relative change in the number of different types of packets.

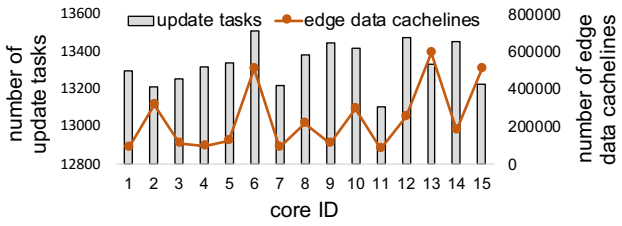


Figure 19: Work distribution among cores in HAU

Impact of other data structures: The baseline (i.e., without RO) of some reordering-friendly cases performs better with Degree-Aware Hashing (DAH) than with Adjacency List (AS). However, for these cases, the performance of AS with batch reordering is on par with the performance of DAH (e.g., for *wiki*-100K, AS with batch reordering provides 1.8× speedup over AS, whereas DAH provides 1.95× speedup over AS. Note that AS with batch reordering and search coalescing provides 2.1× speedup and outperforms DAH performance). However, applying batch reordering degrades the

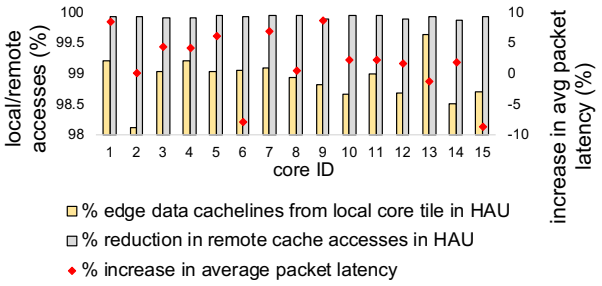


Figure 20: Remote cache accesses and NOC performance

performance of the remaining cases (e.g, *lj*, *patents*, etc.) where AS is the best data structure. Hence, we propose ABR which helps a system maintain only one data structure with an effective adaptive technique based on input characteristics.

Discussion of Recent Streaming Graph Frameworks: This paper focuses on optimizing batch reordering (RO) for streaming graphs based on input knowledge. Recent streaming graph frameworks such as GraphOne [42] and Aspen [26] use RO among other features that further improve streaming graph performance. GraphOne [42] uses RO (referred to as edge sharding) in addition to dual versioning, multi-level storage for edge and adjacency lists, optimized memory allocation based on cachelines, and data access at two different granularities. Aspen [26] uses a new compressed functional tree data structure (C-tree) where batch reordering is performed during updates, in addition to data compression (to optimize storage) and the use of flat snapshots (to optimize edge access latency). Implementing all these optimizations as part of our baseline will have substantial impact on streaming graph performance which will change our speedup results. For example, dual versioning in GraphOne would overlap some of the update latency with compute latency. Aspen's focus is to provide concurrent computations and graph updates. Unfortunately, since these optimizations target both the update and compute phases and overlap their latencies, it is hard to isolate their impact on just the update phases of streaming graphs. For example, Aspen's results on BFS [26] shows a 3% latency increase for concurrent updates and computes compared to latency with no updates. So just optimizing the update phase (our focus in this paper) may have a lower impact on performance in this case. However, our mechanisms would still provide significant speedups in cases where updates are dominant, or for graph algorithms and applications where computations cannot execute concurrently with updates. Furthermore, optimizations in prior work did not consider input knowledge to improve update performance. We focus on the batch reordering optimization in this work to gain insight about the impact of input knowledge.

7 CONCLUSION

We propose input-aware software and hardware solutions to improve the performance of streaming graph workloads. Evaluated across 260 workloads, our proposed techniques provide on average 4.55× and 2.6× speedup in graph update for different input types (on top of eliminating the performance degradation from input-oblivious batch reordering). The graph compute performance is improved by 1.26× (up to 2.7×).

ACKNOWLEDGMENTS

We thank the anonymous reviewers and the anonymous shepherd for their valuable feedback. This work was supported in part by NSF 1816833.

REFERENCES

[1] [n. d.]. https://www.darpa.mil/program/hierarchical-identify-verify-exploit.
[2] [n. d.]. Laboratory for Web Algorithms. http://law.di.unimi.it/datasets.php.
[3] 2017. https://www.boost.org/doc/libs/1_67_0/libs/sort/doc/html/sort/parallel/parallel_stable_sort.html.
[4] 2019. DARPA ERI: HIVE and Intel PUMA Graph Processor. https://fuse.wikichip.org/news/2611/darpa-eri-hive-and-intel-puma-graph-processor/.
[5] 2020. https://graphchallenge.mit.edu/darpa-hive.
[6] 2020. https://software.intel.com/en-us/node/506191.
[7] Masab Ahmad, Halit Dogan, Christopher J Michael, and Omer Khan. 2019. Heteromap: A runtime performance predictor for efficient processing of graph analytics on heterogeneous multi-accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 268–281.
[8] Masab Ahmad and Omer Khan. 2016. Gpu concurrency choices in graph analytics. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.
[9] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2016. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News* 43, 3 (2016), 105–117.
[10] Sam Ainsworth and Timothy M. Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2925426.2926254
[11] Vignesh Balaji and Brandon Lucia. [n. d.]. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 203–214.
[12] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 373–386. https://doi.org/10.1109/HPCA.2019.00051
[13] Abanti Basak, Jilan Lin, Ryan Lorica, Xinfeng Xie, Zeshan Chishti, Alaa Alameldeen, and Yuan Xie. 2020. SAGA-Bench: Software and Hardware Characterization of Streaming Graph Analytics Workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
[14] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).
[15] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.
[16] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Graph. *SIGIR Forum* 42, 2 (2008), 33–38.
[17] Anna D Broido and Aaron Clauset. 2019. Scale-free networks are rare. *Nature communications* 10, 1 (2019), 1–10.
[18] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2018.8547541
[19] György Buzsáki and Kenji Mizuseki. 2014. The log-dynamic brain: how skewed distributions affect network operations. *Nature Reviews Neuroscience* 15, 4 (2014), 264–278.
[20] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating real-time graph mining. In *Proceedings of the fourth international workshop on Cloud data management*. ACM, 1–8.
[21] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 52:1–52:12.
[22] Meeyoung Cha, Alan Mislove, and Krishna P. Gummadi. 2009. A Measurement-driven Analysis of Information Propagation in the Flickr Social Network. In *In Proceedings of the 18th International World Wide Web Conference (WWW'09)*. Madrid, Spain.
[23] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 85–98.
[24] Guohao Dai, Tianhao Huang, Yu Wang, Huazhong Yang, and John Wawrzynek. 2019. HyVE: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing. *IEEE Trans. Comput.* 68, 8 (2019), 1131–1146.
[25] William James Dally and Brian Patrick Towles. 2004. *Principles and practices of interconnection networks*. Elsevier.
[26] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency Graph Streaming Using Compressed Purely-functional Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 918–934. https://doi.org/10.1145/3314221.3314598
[27] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.
[28] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time. In *Proceedings of the 2018 World Wide Web Conference (WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1775–1784. https://doi.org/10.1145/3178876.3186183
[29] Young-Ho Eom and Hang-Hyun Jo. 2015. Tail-scope: Using friends to estimate heavy tails of degree distributions in large-scale complex networks. *Scientific reports* 5 (2015), 09752.
[30] Dhivya Eswaran, Christos Faloutsos, Sudipto Guha, and Nina Mishra. 2018. Spotlight: Detecting anomalies in streaming graphs. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1378–1386.
[31] Priyank Faldu, Jeff Diamond, and Boris Grot. 2020. Domain-Specialized Cache Management for Graph Analytics. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 234–248.
[32] Guanyu Feng, Zixuan Ma, Daixuan Li, Xiaowei Zhu, Yanzheng Cai, Wentao Han, and Wenguang Chen. 2020. RisGraph: A Real-Time Streaming System for Evolving Graphs. *arXiv preprint arXiv:2004.00803* (2020).
[33] Guoyao Feng, Xiao Meng, and Khaled Ammar. [n. d.]. DISTINGER: A distributed graph data structure for massive dynamic graph processing. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 1814–1822.
[34] Oded Green and David A Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.
[35] Ajeet Grewal, Jerry Jiang, Gary Lam, Tristan Jung, Lohith Vuddemarri, Quannan Li, Aaditya Landge, and Jimmy Lin. 2018. Recservice: Distributed Real-time Graph Processing at Twitter. In *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'18)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=3277180.3277183
[36] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.
[37] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 1.
[38] Keita Iwabuchi, Scott Sallinen, Roger Pearce, Brian Van Essen, Maya Gokhale, and Satoshi Matsuoka. 2016. Towards a distributed large-scale dynamic graph data store. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 892–901.
[39] Anand Iyer, Li Erran Li, and Ion Stoica. 2015. Celliq: Real-time cellular network analytics at scale. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 309–322.
[40] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. ACM, 5.
[41] Wole Jaiyeoba and Kevin Skadron. 2019. GraphTinker: A High Performance Data structure for Dynamic Graph Processing. In *2019 IEEE International Parallel Distributed Processing Symposium (IPDPS)*.
[42] Pradeep Kumar and H Howie Huang. 2019. GraphOne: A data store for real-time analytics on evolving graphs. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 249–263.
[43] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[44] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.
[45] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture*. 116–128.
[46] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–14.
[47] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *2019 52nd Annual IEEE/ACM International Symposium on*

Microarchitecture (MICRO-52),. IEEE.

[48] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

[49] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 457–468.

[50] Hamza Omar, Masab Ahmad, and Omer Khan. 2017. GraphTuner: An input dependence aware loop perforation scheme for efficient execution of approximated graph algorithms. In *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 201–208.

[51] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 166–177.

[52] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2020. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*.

[53] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[54] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *2020 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53),*. IEEE.

[55] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

[56] A. Samara and J. Tuck. 2020. The Case for Domain-Specialized Branch Predictors for Graph-Processing. *IEEE Computer Architecture Letters* 19, 2 (2020), 101–104. https://doi.org/10.1109/LCA.2020.3005895

[57] Albert Segura, Jose-Maria Arnau, and Antonio González. 2019. SCU: a GPU stream compaction unit for graph processing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 424–435.

[58] Dipanjan Sengupta and Shuaiwen Leon Song. 2017. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In *International Supercomputing Conference*. Springer, 97–119.

[59] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. Graphin: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*. Springer, 319–333.

[60] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating dynamic graph analytics on gpus. *Proceedings of the VLDB Endowment* 11, 1 (2017), 107–120.

[61] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: real-time content recommendations at twitter. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1281–1292.

[62] Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. GraPU: Accelerate Streaming Graph Analysis Through Preprocessing Buffered Updates. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/3267809.3267811

[63] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 417–430.

[64] Shreyas G Singapura, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K Prasanna. 2017. OSCAR: Optimizing SCrAtchpad reuse for graph processing. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[65] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.

[66] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. 2009. On the Evolution of User Interaction in Facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*.

[67] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic analysis of evolving graphs. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 32.

[68] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 237–251.

[69] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, et al. 2019. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 615–628.

[70] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 544–557.

[71] Jinhong Zhou, Shaoli Liu, Qi Guo, Xuda Zhou, Tian Zhi, Daofu Liu, Chao Wang, Xuehai Zhou, Yunji Chen, and Tianshi Chen. 2017. Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 731–734.

[72] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-based Graph Processing. In *2019 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52),*. IEEE.