GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing

Shafiur Rahman
Computer Science and Engineering
University of California, Riverside
Riverside, USA
mrahm008@ucr.edu

Nael Abu-Ghazaleh
Computer Science and Engineering
University of California, Riverside
Riverside, USA
nael@cs.ucr.edu

Rajiv Gupta
Computer Science and Engineering
University of California, Riverside
Riverside, USA
gupta@cs.ucr.edu

***Abstract*—Graph processing workloads are memory intensive with irregular access patterns and large memory footprint resulting in low data locality. Their popular software implementations typically employ either Push or Pull style propagation of changes through the graph over multiple iterations that follow the Bulk Synchronous Model. The performance of these algorithms on traditional computing systems is limited by random reads/writes of vertex values, synchronization overheads, and additional overheads for tracking active sets of vertices or edges across iterations. In this paper, we present GraphPulse, a hardware framework for *asynchronous* graph processing with *event-driven* scheduling that overcomes the performance limitations of software frameworks. Event-driven computation model enables a parallel dataflow-style execution where atomic updates and active sets tracking are inherent to the model; thus, scheduling complexity is reduced and scalability is enhanced. The dataflow nature of the architecture also reduces random reads of vertex values by carrying the values in the events themselves. We capitalize on the update properties commonly present in graph algorithms to *coalesce* in-flight events and substantially reduce the event storage requirement and the processing overheads incurred. GraphPulse event-model naturally supports asynchronous graph processing, enabling substantially faster convergence by exploiting available parallelism, reducing work, and eliminating synchronization at iteration boundaries. The framework provides easy to use programming interface for faster development of hardware graph accelerators. A single GraphPulse accelerator achieves up to 74x speedup (28x on average) over Ligra, a state of the art software framework, running on a 12 core CPU. It also achieves an average of 6.2x speedup over Graphicionado, a state of the art graph processing accelerator.**

***Index Terms*—Graph Processing, Hardware Accelerator, Event-driven Model, Domain-specific Architecture**

I. Introduction

Computation on large graphs is an important computational workload since graph analytics is employed in many domains, including social networks [10], [15], [22], [38], [52], web graphs [58], and brain networks [9], to uncover insights from high volumes of connected data. Iterative graph analytics require repeated passes over the graph until the algorithm converges. Since real-world graphs can be massive (e.g., YahooWeb has 1.4 billion vertices and 6.6 billion edges), these workloads are highly memory-intensive. Thus, there has been significant interest in developing scalable graph analytics systems. Some examples of graph processing systems include GraphLab [31], GraphX [16], PowerGraph [17], Galois [37], Giraph [4], GraphChi [28], and Ligra [49].

Large scale graph processing introduces a number of challenges that limit performance on traditional computing systems. First, *memory-intensive processing* stresses the memory system. Not only is the frequency of memory operations relative to compute operations high, the memory footprints of the graph computations are large. Standard techniques in modern processors for tolerating high memory latency, such as caching and prefetching, have limited impact because irregular graph computations lack data locality. The large memory footprints also lead to *memory bandwidth bottlenecks* and exacerbate the *long access latencies*. Second, the *synchronization overheads* of accessing shared graph states in most computational models are high due to concurrent updates of vertices. Third, the overheads of *tracking active vertices or edges* are substantial. Such tracking is essential as the computation is irregular with varying subset of vertices and edges being active in each iteration. However, due to the large graph sizes, the book-keeping required can also grow to a substantial size. Because of the above overheads, we believe that modern processing architectures are not well suited for graph processing applications at scale.

The end of Dennard scaling restricts the ability of software frameworks to scale performance by utilizing larger processors due to the Dark Silicon effect. This incentivizes the push towards custom hardware accelerators built for specific application domains that can be orders of magnitude more efficient in terms of performance and power. A large portion of the industry workloads are a small set of repetitive tasks that can benefit greatly from specialized units to execute them. The integration of reconfigurable accelerators in the cloud has gained momentum as evinced by the Microsoft Catapult Project [42] and Amazon F1 FPGA instance [3].

Motivated by the above observations, we propose a new scalable hardware graph processing framework called GraphPulse. GraphPulse alleviates several performance challenges faced in traditional software graph processing, while bringing the benefits of hardware acceleration to graph computations [19]. GraphPulse centers around the idea of *event-driven* computation. It expresses computations as *events*, typically generated when the value of a vertex changes to update

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00078

TABLE I
COMPARISON ON GRAPH PROCESSING MODELS.

	PULL	PUSH	GraphPulse	:	Features
Random Reads	High		Low	:	Events Carry Data
Random Writes		High	Low	:	Event Coalescing + Incremental Algs
Parallelism Scope	BSP Iteration	BSP Iteration	Round	:	Extends Across Multiple Iterations
Synchronization	Global Barrier	Global Barrier	None Needed	:	Asynchronous Algorithms
Tracking Active Sets	Of Vertices Needed	Of Edges Optional	Not Needed	:	Events Represent Active Sets
Atomic Vertex Updates	None Atomic	All Atomic	None Atomic	:	Event Scheduling

vertices on all outgoing edges. Managing event generation and communication to outgoing neighbors incurs substantial overheads in software frameworks. Different techniques, such as *bucketing* in Julienne [14] and GraphIt [63] or *topology aware priority scheduling* in Galois [37], are employed in software to reduce the overhead of managing and ordering the contributions from vertices. By supporting data carrying events as hardware primitives, and routing them within the accelerator, these overheads are largely eliminated, making the GraphPulse model more efficient than software implementations. Previous works showed promising results with similar hardware routing techniques in transactional memory accelerators [11], graph accelerators [1], DES accelerators [33], [43], [44], hardware message passing [40] etc.

Table I summarizes the overheads of the Push and Pull models for graph processing as well as the features in our approach to mitigate them. Since events carry incoming vertex contributions, one of the primary sources of the random memory accesses is eliminated such that memory accesses are only necessary during vertex updates. Moreover, synchronization is simplified by having the accelerator serialize events destined to the same vertex; thus, synchronization overhead of traditional graph processing is reduced. At any point in time, the events present in the system naturally correspond to the active computation and thus the bookkeeping overhead of tracking the active subset of the graph is also masked. Finally, scalability to handle large graphs is achieved by partitioning larger graphs into slices that are processed by the accelerator one at a time (or simultaneously using multiple accelerators though this solution is not explored in this paper). Additional background on conventional graph processing is given in Section II.

Many graph algorithms are iterative where each iteration processes an updated set of active vertices till the termination. For example, a PageRank iteration updates the ranks of active vertices and propagates them to neighboring vertices that are updated in the following iteration. The computation terminates when the updates of rank values become smaller than a set threshold. Many important classes of graph algorithms are amenable to independent updates, i.e. updates arriving at a vertex along one incoming edge can be processed independent of updates arriving at the same vertex along other incoming edges. This property enables the algorithms to execute asynchronously across multiple iterations, without having to orchestrate iterations via synchronizing barriers. The above update property also allows us to *coalesce* in flight events aimed at the same vertex and thus reduce the event storage and processing overheads incurred. The event-based model in GraphPulse naturally supports asynchronous graph processing, achieving substantial performance benefits due to increased parallelism and faster convergence [56], [62].

It becomes readily apparent that, when an event is generated for each outgoing edge of every updated vertex, the event population will soon overwhelm the on chip memory resources and thus necessitate expensive spills to memory. To address this problem, we develop novel queuing structures that coalesce all events aimed at the same vertex into a single event by exploiting the independent update property of the applications. Section III overviews some of the important issues underlying event-based processing, while Section IV presents the GraphPulse design. We also introduce additional optimizations to prefetch the outgoing edge data and to parallelize the generation of the outgoing events in Section V. With these optimizations, GraphPulse is able to outperform Ligra on a 12-core Xeon CPU by up to a factor of 74× (28× on average), and Graphicionado [18], a state of the art graph processing accelerator by 6.2× on average for our benchmarks. Analyzing its energy consumption, we see that it achieves 280× better energy-efficiency than the software framework. Our experimental study and evaluation results are presented in Section VI.

GraphPulse is related to a number of prior works. The architecture operates on unordered data-driven algorithms with local computation operators and unstructured topology [41]. The asynchronous execution model uses an autonomous scheduling approach where the scheduling of active vertices is not restricted by any specific constraint. Other autonomously scheduled data-driven graph-analytics implementations, such as Galois, require priority scheduling or transactional semantics (commit/rollback) for work-efficient parallelism [37]. Chronos [1], a hardware accelerator capable of handling asynchronous graph-analytics, also assigns virtual priority ordering (*timestamps*) on memory read-write objects and uses hardware-assisted commit mechanism to extract massive parallelism speculatively while ensuring transaction safety. Ozdal et al. [39] implemented an asynchronous graph analytics accelerator using a reorder-buffer-like synchronization unit to detect potential conflicts in hardware and serialize vertex execution in presence of possible RAW and WAR hazards. The complexity of such serialization can potentially limit the parallelization opportunities in many applications. GraphPulse, on the other hand, can avoid serialization or ordering because transaction safety is implicitly guaranteed by a coalescing queue. The specialized execution model and the nature of the targeted *delta-accumulative* algorithms [61] let the accelerator schedule the active vertices without any mutual dependence.

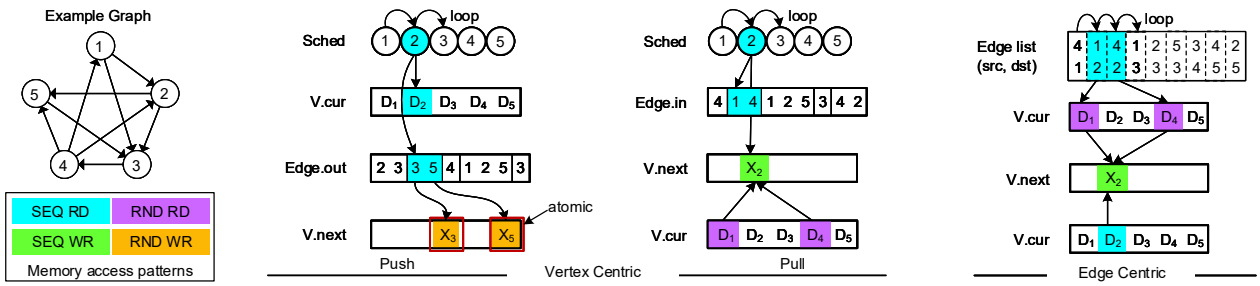

Fig. 1. Data access patterns: Vertex Ordered (*Push* and *Pull* directions) and Edge Centric processing paradigms.

We exploit this flexibility to facilitate locality and memory bandwidth utilization.

The key contributions of this paper are as follows:

- We propose an *event-based* graph processing accelerator, GraphPulse, that addresses many inefficiencies in traditional graph processing frameworks. In addition, GraphPulse performs asynchronous processing of graphs that can result in substantial speedups due to greater exploitation of parallelism and faster convergence.
- We optimize the model by *coalescing* events to control event population and achieve efficient memory access patterns that enable implementation in reconfigurable hardware or ASIC.
- We enhance the model with prefetcher and streaming scheduler to achieve high throughput. This design substantially outperforms software frameworks due to its efficient memory usage and bandwidth utilization.

II. Background and Motivation

Extracting massive parallelism is key to obtaining higher performance on large graphs. However, it is challenging to build an efficient parallel graph processing application from the ground up. Software graph processing frameworks solve this issue by providing simple primitives to the user for describing the algorithm specific operations and relying upon runtime system for complex data management and scheduling. Decoupling the application logic from low level management exposes opportunities to integrate many optimization techniques that are opaque to the application programmer. However, software frameworks do not fully address locality challenges originating from the irregular memory access patterns and synchronization requirements of graph applications.

A. Conventional Computation Models

Graph processing frameworks typically follow either *Vertex-Centric* or *Edge-Centric* paradigm for sequencing their computation. In these frameworks, graph memory contains three components: 1) a vertex property memory containing the vertex attributes; 2) a graph structure specifying the relationships, i.e. edges; and optionally 3) memory for intermediate states of computation in progress.

The scheduling determines the order in which the vertex or structural properties in memory are accessed. The memory access patterns for various approaches are shown in Figure 1. In the *vertex-centric* paradigm the vertex computation performed is designed from the perspective of a vertex, i.e. vertex property value is updated by a computation based upon property values of its neighbors [32]. Most vertex-centric computation models follow one of two approaches: *pull* or *push*. In the *pull* approach, each vertex reads the properties of all its incoming neighbors and updates its value. Thus, it involves random reads; many of which are redundant as the vertex values read may not have experienced any change and hence do not contribute any change to their outgoing neighbors. These redundant reads lead to poor utilization of memory bandwidth and wasted parallelism due to memory latency. On the other hand, *push* approach performs a vertex *read-modify-update* operation for each of its outgoing neighbor. These updates must be performed via atomic operations. Since graph processing algorithms suffer from poor locality resulting in frequent cache misses, atomic operations are very inefficient. For example, a Compare-And-Switch (CAS) operation on an Intel Haswell processor is more than 15 times slower when data is in RAM vs in L1 cache [48].

In an *edge-centric* model, the edges are sorted, typically in the order of their destination vertex ids, and streamed into the processor. The processors read both source and destination to perform the vertex update. This approach either suffers from redundant reads of inactive source vertices or locking overhead of destination vertices. The memory traffic for reading edges of a vertex v typically achieves high spatial locality since the edges are stored in consecutive locations. However, vertex accesses have poor spatial locality as v can be connected to other vertices that are scattered in memory; there is a little chance of vertices being stored in consecutive memory locations. Thus, vertex traffic suffers significantly due to memory access latency being on the critical path. Additionally, since the graphs are large, the reuse distance of a cached memory is also large, i.e. temporal locality is non-existent. Thus, on-chip caches are mostly ineffective and compute resources are poorly utilized.

Without maintaining active sets, many vertices will be read unnecessarily as their values would not have changed in prior iterations. One could simply process all vertices in each iteration and forego the need for maintaining active sets, but this is extremely wasteful because the number of vertices that are active can vary greatly from iteration to iteration. To avoid processing of all vertices, software frameworks typically invest in tracking the *active set* of vertices. While this tracking eliminates redundant processing, it unfortunately

incurs significant overhead for maintaining the active set in the form of bitvector or a list.

Efficient tracking of active set in hardware accelerators is difficult to achieve. The inherent simplicity of the vertex-ordered scheduling is lost due to scheduling and synchronization overheads in the hardware. Additionally, the efficacy of many performance-optimizing hardware primitives is reduced due to the irregularities introduced by active set scheduling.

Efficient handling of vertex updates: Since vertex updates are a crucial bottleneck in a graph-analytics application, some prior works focus on improving the locality and cost of scattering updates to the neighbors. Beamer et al. [8] uses *Propagation Blocking* to accumulate the vertex contributions in cache-resident bins instead of applying them immediately. Later, the contributions are combined and then applied, thus eliminating the need for locking and improving spatial locality. *Propagation Blocking* technique creates perfect spatial locality but fails to utilize potential temporal locality for vertices having many incoming updates since updates are binned and spilled to memory first. Other methods exploit commutative nature of the reduction (apply) operation seen in many graph algorithms to relax synchronization for atomic operations. Coup [60] extends the coherence protocol to apply commutative-updates to local private copies only and reduce all copies on reads. This reduces read and write traffic by taking advantage of the fact that commutative reduction can be unaware of the destination value. PHI [35] also uses the commutativity property to coalesce updates in private cache and incorporates update batching to apply scatter updates in bulk while reducing the on-chip traffic further. Both PHI and Coup optimize memory updates, but have no fine grain control over the memory access pattern. Like PHI, GraphPulse utilizes commutative property to fully coalesce updates using the specialized on-chip queue. Furthermore, GraphPulse applies these updates at a dataflow-level abstraction to reorder and schedule updates for maximizing spatial locality and bandwidth use.

B. Delta-based Accumulative Processing

GraphPulse targets graph algorithms that can be expressed as a delta-accumulative [61] computation – this includes many popular graph processing workloads [20], [56], [57], [61], [62]. In this model, updates aimed at a vertex by different incoming edges can be applied independently. A vertex whose value changes, conveys its *"change"* or *delta* to its outgoing neighbors. The neighbors update themselves upon receiving the *delta*, and propagate their own delta further. Thus, the computation is turned into a data flow computation that remains active as long as necessary until convergence. The continuous tracking of the active set is inherent to the data flow model. The updates are broken into two steps:

$$\begin{cases} v_j^k & = & v_j^{k-1} \oplus \Delta v_j^k \\ \Delta v_j^{k+1} & = & \sum_{i=1}^{n} \oplus g_{\langle i,j \rangle} \left(\Delta v_i^k \right) \end{cases} \quad (1)$$

v_j is the vertex state. Δv_j is the change to be applied to the vertex using algorithm specific operator '$\oplus$'. The two equations can be visualized as a sequence of recursive operations going back to the initial conditions v_j^0 and Δv_j^0 that are also specific to the algorithm under consideration. We highlight two key components in the equation: $g_{\langle i,j \rangle}$, the *propagate* function, which modifies and conveys the change(*delta)* in the vertex value to its neighbors; and '$\oplus$', the *reduce* function, that both reduces the propagated *deltas* to compute new *delta* and applies it to the current vertex state. To express an iterative graph algorithm in the incremental form, we make use of the following two properties:

Reordering Property. The *deltas* can be applied to the vertex state in any order. This reordering is allowed when the propagation function $g_{\langle i,j \rangle}$ is distributive over $\oplus$, i.e., $g(x \oplus y) = g(x) \oplus g(y)$; and $\oplus$ is both commutative and associative, i.e., $x \oplus y = y \oplus x$ and $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.

Simplification Property. Given an edge $i \rightarrow j$, the '$\oplus$' operation is constructed to incrementally update the vertex value v_j when there is a change in v_i. Therefore if v_i does not change, it should have no impact on v_j. That is,

$$v_j^k \oplus g_{\langle i,j \rangle} \left(\Delta v_i^k \right) = v_j^k \text{ if } \Delta v_i^k = 0$$

This property is satisfied when $g_{\langle i,j \rangle}(\cdot)$ is constructed to emit an *Identity* value for the reduce operator $\oplus$ when $\Delta v_i = 0$.

A wide class of graph algorithms – PageRank, SSSP, Connected Components, Adsorption, and many Linear Equation Solvers – satisfy the above properties [61]. However, there are exceptions. For example, graph coloring cannot be expressed since the update is a function of all vertex values obtained along the incoming edges, i.e. they cannot be updated using a value obtained along a single edge. Delta-based update algorithms break the iteration abstraction, allowing asynchronous processing of vertices and thus substantially increasing available parallelism, removing the need for barrier synchronization at iteration boundaries (required by the *Bulk Synchronous Parallel* Model [55]), and providing opportunities for combining multiple delta updates. All these properties are exploited by GraphPulse to improve performance.

III. GraphPulse Design Overview

Before introducing the GraphPulse accelerator architecture, we overview important considerations in the event processing model (see Algorithm 1). This section also discusses the mapping of a delta-based graph computation to GraphPulse.

A. Event-Processing Considerations

Computation with Delta/Data Carrying Events. In the delta-based model, the only data that is passed between vertices are the *delta* messages. These messages (implemented as events) encode the computation and carry the input data needed by the computation as well, removing the need for expensive reads of the input set of a vertex computation. Moreover, vertex updates can be performed asynchronously; in other words, a vertex can be updated at any time with the *delta* it has received so far. Based on these two properties, we develop an event-driven model to support delta-based graph computation. This approach completely decouples the communication and control tasks of the graph computation.

Algorithm 1: Event-Driven Graph Processing Model for Incremental PageRank

Data: Graph Structure $G(V,E)$
Result: Vertex Properties V

$V[:] \longleftarrow InitialVertexProperty()$
$Queue \longleftarrow InitialEvents(\{v_i \in V\})$
while *Queue is not* empty **do**
  $(u, \delta) \longleftarrow pop(Queue)$
  $temp \longleftarrow V[u]$
  $V[u] \longleftarrow \underline{Reduce}(V[u], \delta)$
  $\Delta_u \longleftarrow (V[u] - temp)$
  if $abs(\Delta_u) >$ `THRESHOLD` **then** `// (Term. cond.)`
    foreach *outgoing edge* $E_{u,v}$ *of vertex* u **do**
      $\delta_v \longleftarrow \underline{Propagate}(\Delta_u, E_{u,v})$
      if (w, δ_w) *exists in Queue where* $w = v$ **then**
        $\delta_w \longleftarrow \underline{Reduce}(\delta_w, \delta_v)$ `// Coalesce`
      else
        $Queue \longleftarrow insert(v, \delta_v)$
return V

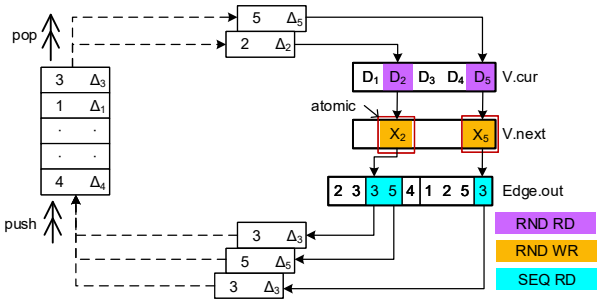

Fig. 2. Data access pattern in event-driven approach with a FIFO event-queue.

We define an *event* as a lightweight-message that carries a *delta* as its payload. Multiple events carrying *deltas* to the same vertex can be combined using a *reduce* operator specific to the application to reduce the event population and, subsequently, event storage and processing overheads. Execution of a vertex program can only be triggered by an event. A set of initial events is created at the beginning as part of the application initialization. When a processor receives an event for a vertex, it executes two tasks: 1) *update* of the vertex state using the *reduce* operation, and then 2) *generate* a new set of events using the *propagate* function described in Section II-B. The newly generated events are collected in an event queue from which they are scheduled to other processors to start execution of new vertex programs.

Figure 2 shows a view of the computation model. At any time, the event queue has a set of pending events. The event at the head of the queue is issued to a processor which read-modify-writes the vertex value, then reads the corresponding adjacency list to prepare and insert new events into the event queues. The memory accesses are still in random order and require locking for parallel operation since two or more events to the same vertex may be issued from the queue; our optimized design mitigates both of these limitations.

Coalescing Inflight Events. As discussed in Section II-B, the reordering property of the propagation parameter allows the architecture to combine multiple events to the same destination while still in the queue using the *reduce* function without affecting program correctness; we call this operation *event coalescing*. Event coalescing is critical for a practical asynchronous design because every event in the queue can cause the generation of new events for every outgoing neighbor or destination vertex, unless a termination condition is met. Consequently, for every event consumption, new events are produced and the number of events in the system will rapidly grow. For designing an event-driven processor with limited storage, we require the event coalescing capability to ensure control over the rate of event generation.

Implicit Atomic Updates. In parallel execution, processors may attempt to update the same vertex's state simultaneously, necessitating locking or atomic updates. In Graphpulse, all the vertex memory accesses are associated with an event, and an event only modifies a single vertex value. With the guarantee that, via coalescing, no more than one event is in-flight for any vertex, safety for atomic access is naturally ensured. Our implementation completely coalesces all events targeted to a vertex into one before it is scheduled preventing race conditions that can otherwise arise in presence of concurrent updates.

Isolating Control Tasks from Computation. All memory accesses to vertex and edge data are isolated to the algorithm specific task processing logic. The control tasks, which primarily consist of scheduling of vertex operations, are naturally encapsulated using the *events* abstraction, and do not require any accesses to the graph data to schedule their computation. Coupled with the guarantee of memory consistency, this isolation makes the vertex scheduling logic extremely simple and the datapath highly independent and parallelizable. Also, the memory interfaces designed are simple and efficient since there are only simple memory accesses to the vertex properties. This model reduces memory accesses compared to the classical graph processing approaches including *Vertex-Centric Push/Pull* and the *Edge-Centric* paradigms.

Active Set Maintenance. The events resident in the queue encapsulate the entire active computation, which provides an alternative way to manage *active sets* using hardware structures. Vertices that are inactive will have no events updating them; and the set of unprocessed events indicate a set of vertices that are to be activated next. Most existing graph frameworks use bitmaps or vertex-lists to maintain an active set which entails significant management overhead. The event maintenance task is decoupled from the primary processing path in our model which results in greater parallelization opportunities. Efficient fine-grained control over the event-flow, thereby the scheduling of vertices, can be achieved via hardware support.

Initialization and Termination. After loading a graph, we bootstrap the computation as follows. We define an *Identity* parameter that, when passed to the *reduce* operator with another value, leaves the latter unchanged (e.g., 0 is *identity* for the *sum()* operation). We set the vertex memory to the identity parameter for the graph. The initial events, that are set with the initial target value of the vertices, populate the event queue. The first event of a vertex is guaranteed to trigger

TABLE II
FUNCTIONS FOR MAPPING ALGORITHM TO GRAPHPULSE

	propagate(δ)	reduce	$V_{j,init}$	$\Delta V_{j,init}$
PR-Delta	$\alpha \cdot E_{i,j} \cdot \delta / \mathtt{N(src)}$	+	0	$1-\alpha$
Adsorption	$\alpha_i \cdot E_{i,j} \cdot \delta$	+	0	$\beta_j \cdot I_j$
SSSP	$E_{i,j} + \delta$	min	∞	0 (j=r); ∞
BFS	0	min	∞	0 (j=r); ∞
Conn. Comp.	δ	max	-1	j

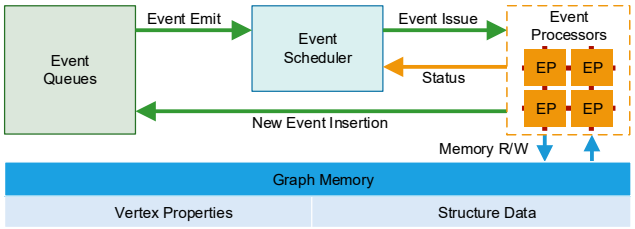

Fig. 3. Overview of GraphPulse Design

a change and then propagate it to other vertices to bootstrap the computation. The event population eventually declines as the computation converges; an update event may not generate new events if the update magnitude is below a threshold (e.g., in PageRank). Eventually, the event queue becomes empty without replenishment and the application terminates when there is no more event to schedule. We also provide the ability to specify a global termination condition for better control with some applications (see Section IV-C).

B. Application Mapping

To implement a delta-based accumulative graph computation in our model shown in Algorithm 1, the user must define several functions described next – Table II shows the reduction and propagation functions, and the initialization values for five graph applications.

`Reduce` function expresses the *reduction* operation that accumulates incoming neighbors' contributions to a vertex, and coalesces event *deltas* in queue. It takes a delta value and current vertex state to update $state = state \oplus delta$.

`Propagate` function expresses the source vertex's *propagation* function ($g(x)$) that generates contributions for the outgoing neighbors. It uses the change in state to produce outgoing *delta*, $\Delta_{out} = g_{\langle E.src,E.dst \rangle}(\Delta V)$.

`Initialization` function defines the initial vertex states to an *identity* value for the reduction operator. Also, the initial event delta is set such that `Reduce(Identity, delta)` results in the intended initial state of the target vertex.

`Terminate` function defines a local *boolean* termination condition in the framework that checks for changes in the vertex state. Propagation for an event stops when the local termination condition is valid and the vertex state is unmodified. The program stops if all events have terminated locally.

Programming Interface. Due to the simple programming abstraction, user effort is modest. The user can define program logic in HDL using custom functional modules and pipeline latency, or use some common functional modules in GraphPulse (e.g., *Min, Max, Sum*). The user also creates the array of initial events and vertex states, which are written to the accelerator memory and registers by the host CPU.

IV. GRAPHPULSE ARCHITECTURE

GraphPulse is an event-based asynchronous graph processing accelerator that leverages the decoupled nature of event-driven execution. The event processing datapath exposes the computational parallelism and exploits available memory bandwidth and hardware resources. The accelerator takes advantage of low-latency on-chip memory and customizable communication paths to limit the event management and scheduling overheads. The following insights from Section III-A guide the datapath design:

1) Vertex property reads and updates are isolated and independent, eliminating the need for atomic operations. When sufficient events are available for processing, the throughput is only limited by the memory bandwidth.
2) To sustain many parallel vertex operations, it should be possible to insert, dequeue, and schedule events with high throughput.
3) Since no explicit scheduling is needed, the number of parallel vertex processing tasks can be easily scaled to process increasingly larger graphs.

We next describe a baseline implementation guided by these considerations, and then describe the optimizations we incorporate to improve its performance.

A. Abstractions for Events

Figure 3 overviews the architecture of GraphPulse. The primary components of the accelerator are Event Queues, the Event Scheduler, Event Processors, the System Memory, as well as the on-chip network interconnecting them. The event processors directly access the memory using an efficient high-throughput memory crossbar interface. For scalability our goal is to leverage the bandwidth to support high degree of memory parallelism and simultaneously present many parallel requests to memory. Because events are the unit of computation, we aim to fit all active events in on-chip memory to avoid having to spill and fetch events. However, for larger graphs, this is not possible, and we use a partitioning approach to support them (see Section IV-F). We consider a configuration with 256 event processors for our baseline.

B. Event Management

Event Queue stores the events representing the active vertex set of the graph. Events are stored as a tuple of destination vertex ID and payload (delta). Schedulers drain events from the queue in sequence giving them to the processors, while newly generated events are fed back to the queue. Since events are generated for all edges, the volume of events grows rapidly, which represents an obstacle for efficient processing. Moreover, due to the asynchronous processing, multiple activations of a vertex can coexist that then generate multiple set of events over the vertex edges. Figure 4 shows the total number of

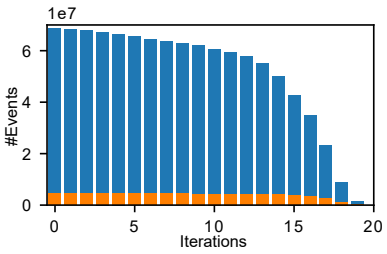

Fig. 4. Total events produced (blue) and remaining after coalescing (orange).

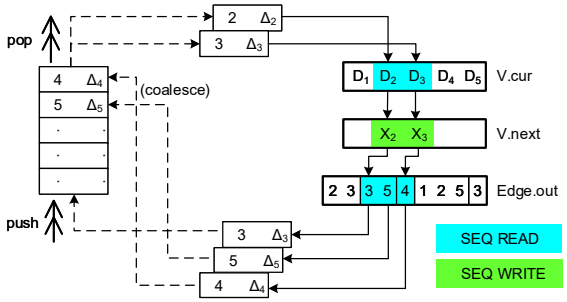

Fig. 5. Data access pattern in event-driven approach with coalescing & sorting.

events produced during each iteration and the number of events remaining after coalescing for PageRank running on the LiveJournal social network graph [6] (∼5M nodes, ∼69M edges). We see that over 90% of the events are eliminated via coalescing multiple events destined to the same vertex. Dramatic reduction in the number of events also reduces the numbers of computations and memory accesses. The queue is modeled as a group of collector bins dedicated to a subset of vertices to simplify and scale event management. We set up the mapping of vertices to bins such that a block of vertices close in memory map to the same bin. Thus, when events from a bin are scheduled, the set of vertices activated over a short period of time are closely placed in memory and thus the memory accesses exhibit high spatial locality. The ordering approach transforms the inefficient random read/writes into sequential read/writes as shown in Figure 5.

Events are deposited in the bin and the coalescer iterates over the events and applies the Reduce methods over matching events. Following coalescing passes, only a small fraction of unique events remain. However, buffering uncoalesced events significantly increases pressure on the event queues, increases congestion and requires large memory. Therefore to address this limitation, in Section IV-D, we present an in-place coalescing queue that combines events during event insertion.

C. Event Scheduling and Termination

The event scheduler dequeues a batch of events in parallel from the collectors. It arbitrates and forwards new events to the idle processors via the interconnection network. Scheduler drains events from one bin at a time, and iterates over all bins in a round-robin manner (other application-informed policies are possible). We call one complete pass over all bins a *round*. The scheduler allocates events to any idle processor through an arbiter network. The processing cycle for an event begins with the event scheduler dequeueing an event from the output buffer of the event queue when it detects an idle processor. The event is sent via the on-chip routing network to the target processor. Upon receiving an event, the processor starts the vertex program that can cause memory reads and updates of the vertex state. After processing the event, the processor produces all output events to propagate update of its state to directly impacted vertices along outgoing edges. The events produced are sent to the event queues mapped to the impacted vertices.

Global Termination Condition. The scheduler maintains an accumulator to store the local progress from the processors after they perform updates. The default behavior is to terminate when no events remain. However, for applications that can propagate events indefinitely, an optional termination condition provides a way to stop the execution based on a user defined condition such as a convergence threshold. For example, PageRank terminates when the sum of changes in score of all vertices are lower than a threshold. Here, processors pass the deltas as local progress updates to the scheduler where they are summed. A pass over the queue means all active vertices are accessed once, and the global accumulator represents global progress, which can be used for termination condition.

D. In-Place Coalescing Queue

To avoid rapid growth of event population, we explore an in-place coalescing queue that combines events during event insertion, compressing the storage of events destined to the same vertex. If no matching event exists, the event is inserted normally. Conversely, if an event exists, we simply combine the deltas based on the reduction function for the application.

We use multiple bins inside the queue, with each bin structured like a direct-mapped cache (Figure 6(a)). The bins are split into rows and columns, and only one vertex ID maps to a bin-row-column tuple so that there is no collision. Vertex ID isn't stored since the events are direct mapped. Vertices are mapped in column-bin-row order so that clusters in the graph are likely to spread over multiple bins. The number of rows is based on the on-chip RAM block granularity (usually 4096) and multiple memory blocks are operated side-by-side to get a wider read/write interface that can hold power-of-two number of columns. Each bin consists of a *Simple Dual-Ported* RAM block (with one read and one write port).

Insertion and Coalescing. Each bin can accept one new event per cycle, but the insertion has multi-cycle latency. Specifically, insertion units are pipelined so that a bin can accept multiple events in consecutive cycles. In the first cycle during the insertion of an event, the event in its mapped block (if one exists) is read using the read port. In the next cycle, the incoming event is pushed into a combiner function pipeline along with the existing event (FPA unit in Figure 6(b)). We use four stages in the pipeline since most common operators can be designed to have less than 4 cycles latency (e.g., 3 cycles for floating point addition) while maintaining desirable clock speed. After the operation is finished, the combined event

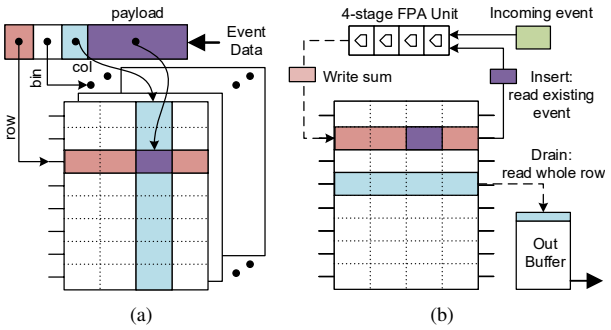

Fig. 6. (a) Direct mapping an event to a queue block; and (b) In-place coalescing and retrieval using direct mapped event storage (for PageRank).

is sent to the write port. During these 4 cycles, other event insertions can be initiated for another row, since the read-write ports are independent and the coalescer is pipelined. When insertions contend for the same row, the later events are stalled until the first event is written.

Removal. Events are removed from one bin at a time in a round-robin fashion. When it's time to schedule events from a bin, a full row is read in each cycle and the events are placed in an output buffer. We prefer wide rows so that many events can be read in one cycle. Insertion to the same bin is stalled in the cycles in which a removal operation is active. Often towards the beginning or the end of an application, the queue is sparsely occupied. It might waste many cycles sweeping over empty rows in these situations. We mark the row occupancy using a bit-vector for each bin. A priority encoder gives fast look-up capability of occupied rows during sweeping the queue.

Due to coalescing at insertion, only one event exists for a vertex in the queue. As removal is done by sweeping in one direction in the bins, we can issue only one event for a vertex in a given round. After a round is complete, the scheduler waits until all the cores are idle before rolling over to the first bin again. This guarantees that race conditions cannot occur without the need for atomic operations or per vertex synchronization.

Another advantage of event coalescing is its ability to combine the effect of propagation across multiple iterations, which is a virtue of the asynchronous graph processing model. Figure 7 shows an example: the delta from processing event A in bin 1 is sent to vertex C mapped to bin 2, where another event for vertex C already exists. Due to coalescing, vertex C will pick up the contribution that otherwise would have been processed in the next iteration. Similarly event E will compound the effect of A two iterations earlier than usual. We call this effect *lookahead*. In Figure 4, we showed that a significant fraction of the events are eliminated by coalescing. Figure 8 shows the degree of lookahead contained in these coalesced events for each round in a 256-bin event queue during PageRank-Delta running on the LiveJournal graph. Because of coalescing and asynchronous execution, an event quickly compounds the effects of hundreds of previous iterations of events in a single round. Note that, the contributions from

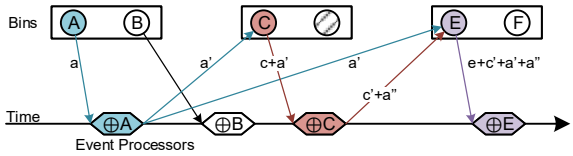

Fig. 7. Compounding of vertex contributions across iterations.

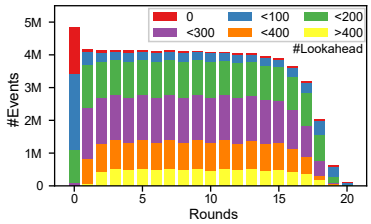

Fig. 8. Degree of lookahead in events processed in each round.

many vertices should quickly stop propagating in uncoalesced model because of damping in PageRank, but they carry on here after being compounded with a bigger valued event. Coalescing exploits temporal locality for the graph, while binning promotes spatial locality, without requiring large caches.

The event-driven approach is prohibitively expensive to implement in software due to the high overhead for generation, management, queueing, sorting, coalescing and scheduling of events using message passing in software. However, since these primitives are directly implemented by the accelerator in hardware, the overheads are essentially mitigated.

E. Event Processors and Routing Network

The event processors are independent, parallel, and simple state machines. The processors are connected to the scheduler using a broadcast network to enable delivery of events from any bin to any available processor. A memory bus connects the event processors to the main memory for reading graph properties. The graph is stored in a Compressed Sparse Row format in memory. The state machine starts after receiving a new event from the scheduler. It reads the vertex property from memory, computes update from the received event using the `reduce()` function, and writes update to the memory in the subsequent steps. It resolves local termination check, and starts reading from *EdgeTable* if it is not terminated. Then, it uses `propagate()` function to compute new delta using the neighbor ID. It pushes the new events to a broadcast channel which connects to the event queues where they are picked up. After finishing its tasks, the processor generates a local progress update (also defined by the application) that is passed to the scheduler along with the processor's status message for global progress checking. In our evaluation, we assumed that event processing logic is specified via a Hardware Description Language, resulting in specialized processors for the application. However, the function encapsulation provides a clean interface to build customizable event processors or use a minimalistic CPU for the event processors.

The baseline GraphPulse configuration consists of 256 processors on a system connected to 4 DRAM memory controllers and coalescing event queues. The scheduler-to-processor interconnect for the baseline design is a multi-staged arbiter network. The processor-to-queue network is a 16x16 crossbar with 16 processors multiplexed into one crossbar port. The complexity of the network is minimized by a number of characteristics of the design: (1) we only need unidirectional dataflow through the network; (2) the datapath communication can tolerate delays arising due to conflicts enabling us to use multi-stage networks and to share ports among multiple processing elements; and (3) our events are fixed in size so that we do not face complexity of variable size messages. We note that the optimizations in Section 5 allow us to retain performance with a much smaller number of cores which further reduces interconnect complexity.

F. Scaling to Larger Graphs

GraphPulse uses the on-chip memory to store the events in the coalescer queue. Each vertex is mapped to an entry in the coalescer, which puts a limit on the size of the active portion of the graph to be less than the maximum number of vertices serviced by the coalescer. For large graphs, the on-chip memory of the accelerator will, in general, not be big enough to hold all vertices. The inherent asynchronous and distributed data-flow pattern of GraphPulse model allows it to correctly process a portion of the graph at a time. Thus, to handle large graphs, we partition the graph into multiple slices such that each slice completely fits into the on-chip. Each slice is processed independently and the events produced from one slice are communicated to other slices. This can be achieved using two different strategies: a) on-chip memory can be shared by different slices sequentially over time while the inter-slice events are temporarily stored in off-chip memory; and b) multiple accelerator chips can house all slices while an interconnection network streams inter-slice events in real-time. We use the first option to illustrate GraphPulse scalability.

We assume that the graph is partitioned offline into slices that each fits on the accelerator [25], [50], [51]. Most graph frameworks employ either a *vertex-cut* or *edge-cut* strategy in partitioning graphs. Since our model is dependent on the number of vertices, we limit the maximum number of vertices in each slice while minimizing edges that cross slice boundaries. We relabel the vertices to make them contiguous within each slice. When a slice is active, the outbound events to other slices are stored in off-chip memory. These events are streamed in later when the target slice is swapped in and activated. Partitioning necessarily gives rise to increased off-chip memory accesses and bandwidth demand. However, the events do not require any particular order for storing and retrieval. We buffer the events that are outbound to each slice to fill a DRAM page with burst-write. When a slice is marked for *swap-out*, the bins are drained to the buffer and the new active slice's events are read in from memory. Both the read and write accesses to the off-chip memory is very fast since they are sequential and can be done in bursts. The bins in the queues have their independent

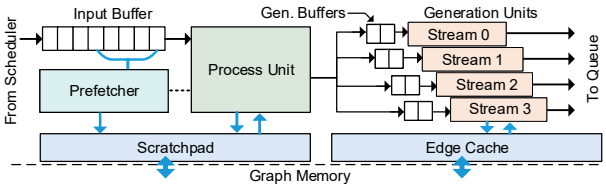

Fig. 9. Optimization of event processing and generation.

pipelined insertion units that can insert the *swapped-in* events in parallel without delay. Event coalescing occurs during insertion of events into the bins. Normal operation can proceed as soon as the first bin is swapped-in, allowing the swap-in/swap-out process to be pipelined, and masking the *switch-over* latency.

V. GraphPulse Optimizations

In this section, we discuss optimizations and extensions to the baseline GraphPulse. Analyzing the performance of the event execution, we discovered that the event processing time was dominated by two overheads: (1) memory accesses to obtain the output vertices needed to identify the targets of the generated events; and (2) the sequential cost to generate the outgoing events. In this section, we introduce two optimizations to alleviate these overheads.

Prefetching: Graph processing applications have notoriously low computation to memory latency ratio. We implement a prefetching scheme to prevent starvation and idling of the event processors. An input buffer is added to the processors and a small scratchpad memory sits between the processor and the graph memory to prefetch and store vertex properties for the events waiting in the input buffer as shown in Figure 9. Prefetching is possible since we know the vertices mapped to each coalescing queue, and we are able to accurately prefetch their outgoing set while they are scheduled for execution. We map the events in the queue such that a block of vertices that are adjacent in graph memory remains adjacent in the queue. The events in a block are *swept* and streamed together to the same input buffer. The predictor inspects a window from the buffer to prefetch only the required data in *cache-line-size* granularity. A carefully sized block (128 in this work) will cause prefetch of all required addresses from a DRAM page together, allowing higher bandwidth utilization than possible via caching alone. Since processors no longer manage data themselves and the memory latency is separated from their critical path, we employ fewer processors (8 in the experiments) to process only the vertices with data available for processing.

We include a small caching buffer with the edge memory reader to enhance the throughput. Prefetching the outgoing edges makes it possible to streamline the generation of events without experiencing expensive memory accesses during event generation. This substantially reduces the event processing time and enhances the event processing throughput. A simple N-block prefetching (N=4) scheme is used for edge memory reads. Since the degree of a vertex are known during the processing phase, we pass this information to the generation unit encoded in the vertex data as a hint for the edge prefetcher to set the

limit of prefetching (N) to avoid unnecessary memory traffic for low degree vertices.

Efficient Event Generation: The memory traffic requirement for edge data compared to vertex properties is very high for most graphs: edge data is typically orders of magnitude larger for most graphs. After an event is processed, update events are generated to its outgoing edge set. We observed that this step is expensive and frequently stalls the event processors limiting processing throughput. The data per edge is small (4 bytes in most of our graphs and applications). This makes reading and generation of events for multiple edges in the same cycle essential for saturating memory bandwidth. Since the data dependence between the processing and event generation phase is unidirectional, we decouple the processor into two units: Processing and Generation (see Figure 9). We increase the event generation throughput by connecting multiple of these generation streams to the same processing unit. A group of streams in one generation unit share the same cache but multiple ports in the event delivery crossbar. Each generation stream is assigned one vertex from the processing unit when idle. Thus, we use parallelism to match the event generation bandwidth to the event processing bandwidth enabling the processing units to work at or near capacity.

VI. Experimental Evaluation

Next we evaluate GraphPulse along a number of dimensions: performance, memory bandwidth requirements, hardware complexity, and power consumption. First we describe our experimental methodology.

A. Experimental Methodology

System Modeling. We use a cycle accurate microarhitectural simulator based on Structural Simulation Toolkit [45] to model the primary components, the memory controller, and interconnection network. The event processor models are designed as state machines with conservative estimation for latency of the computation units. The memory backend is modeled with DRAMSim2 [46] for realistic memory access characteristics. The coalescing engine was modeled as a 4 stage pipelined floating point unit in RTL. The interconnection network is simulated with input and output queue to ensure congestion does not create a bottleneck.

Comparison Baselines. We compare the performance of GraphPulse with a software framework, Ligra [49]. We chose Ligra as the software baseline because, along with Galois [37], it is the highest performing generalized software framework for shared-memory machines [64]. Moreover, Ligra has an efficient shared memory implementation of one of the most robust technique for active set management and versatile scheduling depending on the active set, which is at the core of our work. We considered frameworks that support delta-accumulative processing but those were all targeted for distributed environments and performed much slower than Ligra. We measure the software performance on a 12-core Intel Xeon CPU. The relevant configurations for both systems are given in Table III.

TABLE III
Device configurations for software framework evaluation and GraphPulse with optimizations.

	Software Framework	GraphPulse
Compute Unit	12× Intel Xeon Cores @3.50GHz	8× GraphPulse Processor @ 1GHz
On-chip memory	12MB L2 Cache	64MB eDRAM @22nm 1GHz, 0.8ns latency
Off-chip Bandwidth	4× DDR3 17GB/s Channel	4× DDR3 17GB/s Channel

TABLE IV
Graph workloads used in evaluations.

Graph	Nodes	Edges	Description
Web-Google(WG) [30]	0.87M	5.10M	Google Web Graph
Facebook(FB) [54]	3.01M	47.33M	Facebook Social Net.
Wikipedia(Wk) [13]	3.56M	45.03M	Wikipedia Page Links
LiveJournal(LJ) [6]	4.84M	68.99M	LiveJournal Social Net.
Twitter(TW) [27]	41.65M	1.46B	Twitter Follower Graph

In addition, we compare the performance with a hardware accelerator Graphicionado [18], a state of the art hardware-accelerator for graph processing that uses the Bulk Synchronous execution model. Since the implementation of Graphicionado is not publicly available, we modeled Graphicionado to the best of our ability with the optimizations (parallel streams, prefetching, data partitioning) proposed by the authors. We also gave zero-cost for active vertex management and unlimited on-chip memory to Graphicionado to simplify implementation, making our speedup vs. Graphicionado conservative. We provision Graphicionado with a memory subsystem that is identical to that of GraphPulse.

Workloads. We use five real world graph datasets – Google Web graph, Facebook social network, LiveJournal social network, Wikipedia link graph, and Twitter follower network in our evaluations obtained from the Network Repository [47] and SNAP network datasets [29] (see Table IV). We evaluate five graph algorithms – PageRank (PR), Adsorption(AD), Single Source Shortest Path (SSSP), Breadth-first Search (BFS) and Connected Components (CC) on each of these graphs. We use the contribution based PageRank implementation (commonly referred to as PageRankDelta), which is a delta-accumulative version of PageRank. PageRankDelta execution was faster than the conventional PageRank in the Ligra software framework and Graphicionado for our graph workloads, and therefore we use it for our baselines as well. Ligra does not provide a native Adsorption implementation. We created randomly weighted edges for the graphs and normalized the inbound weights for each vertex. PageRank-Delta model was modified to consider edge weights and propagate based on the functions provided in Table II for Adsorption. Twitter is large and does not fit within the accelerator memory; thus we split it into *three slices* with one slice active at a time using the methodology from

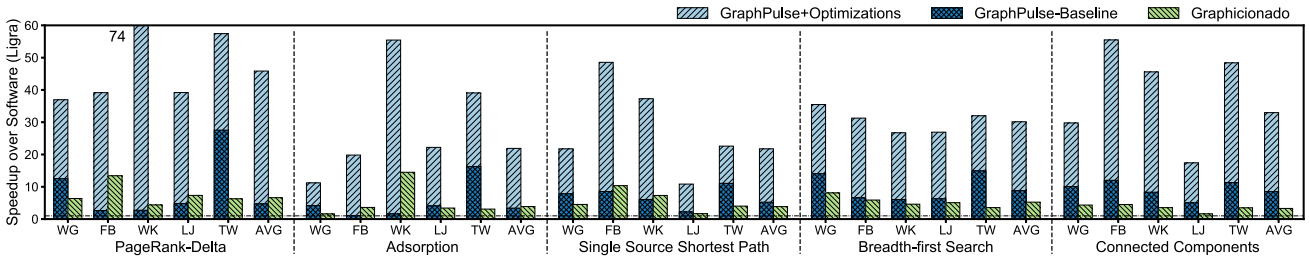

Fig. 10. Performance comparison between GraphPulse, Graphicionado [18], and Ligra [49] framework all normalized with respect to the Ligra software framework. Twitter required partitioning from Section IV-F.

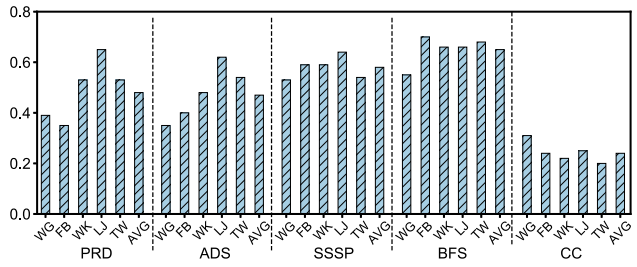

Fig. 11. Total off-chip memory accesses of GraphPulse normalized to Graphicionado.

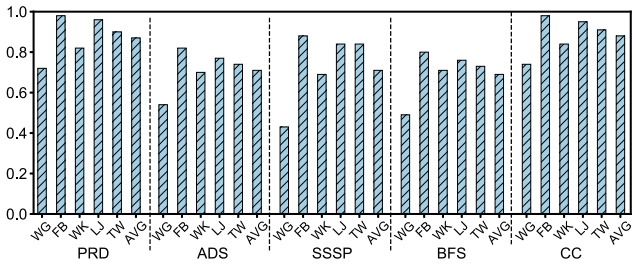

Fig. 12. Fraction of off-chip data utilized.

Section IV-F.

B. Performance and Characteristics

Overall Performance. Figure 10 shows the performance of the GraphPulse architecture in comparison to the Ligra software framework. We observe an average speedup of 28× (10× to 74×) for GraphPulse over Ligra across the different benchmarks and applications. The speedup mainly comes from hardware acceleration, memory friendly access pattern, and the on-the-fly event coalescing capability. BFS, SSSP and CC have similar traversal algorithms. However, BFS and SSSP performance suffers because fewer vertices are active at a time and vertices are reactivated in different rounds of the computation in contrast to CC where the full graph is active for the majority of the computation. The Twitter graph achieves comparable speedup to the other graphs, despite the fact that it incurs the overhead of switching between active slices. Our intuition is that software frameworks incur more overhead for large power law graphs for a computation like PageRank where vertices are visited repeatedly; these overheads are not incurred by GraphPulse as communication is mostly on chip.

Comparing GraphPulse performance to Graphicionado [18], we found that, on average, GraphPulse is about 6.2× faster. The Figure also shows the performance of both the baseline and the optimized version of GraphPulse (with prefetching and parallel event generation); we see that the two optimizations dramatically improve performance.

Memory Bandwidth and Locality. GraphPulse implements a number of optimizations to promote spatial locality and utilize DRAM burst transfer speed whenever possible. Figure 11 shows the total number of off-chip memory accesses required by GraphPulse normalized to Graphicionado. Even compared to the efficient data access of Graphicionado, GraphPulse requires 54% less off-chip traffic on average. GraphPulse 's processing model is memory friendly with events carrying the input data to the computation. Coalescing and *lookahead* also contribute heavily to reduce data traffic by combining computations and memory accesses and stabilizing many nodes earlier. The effect is particularly apparent in CC, where many vertices gets stabilized with the very first event. Finally, Figure 12 shows that in GraphPulse very large fraction of data brought via off-chip accesses is utilized by the computation supporting its ability to reduce random memory accesses.

Event Execution Profile. The average life-cycle of an event is highly dependent on the graph structure and the algorithm. Figure 13 shows a breakdown of average time spent in different stages of the processing path for an event. Individual vertex memory reads have long memory latency. But due to locality aware scheduling and prefetching in the input buffer, latencies for the accesses are masked and the average latency for the vertex memory reads become only few cycles. This indicates the efficiency of the prefetcher. The process stage takes only few cycles too because of pipelining and brevity of typical *apply* tasks. The *Gen Buffer* stage shows the time spent in the input buffer of generation streams after an event is processed and waiting for generation units to be available. The time spent on edge memory access appears to be high, but this is due to the large number of edges that need to be read for event generation in power-law graphs. Figure 14 shows the fractions of time the processors and generators spend accessing memory, processing and stalling. It is noticeable that event

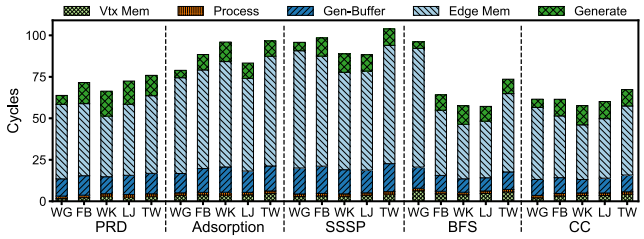

Fig. 13. Cycles spent by an event in each execution stage, shown chronologically from bottom to top.

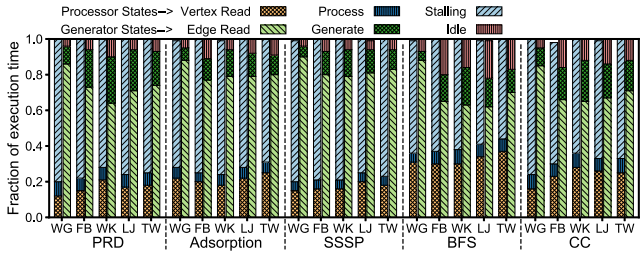

Fig. 14. Time breakdown for the processor (left-bar) and generation units (right-bar).

generation units (right-side bar) spend close to 80% of the cycles reading edge memory. This includes the latency to both read edges from cache and fetch from main memory. We observed that the generation units saturate the memory bandwidth with prefetching and high off-chip utilization. The processors (left hand bars) stall for about 70% of the cycles waiting for generators to become available. We observed that this can be reduced to less than 40% by doubling the ratio of generation streams at the trade-off of increased routing complexity.

C. Hardware Cost and Power Analysis

The coalescing event queue consumes the most power since it closely resembles a cache in design and operation with the addition of a coalescer pipeline. We model the queue using 64MB on-chip memory composed of 64 smaller bins operating independently. 8 scratchpads with 1KB capacity are placed alongside 8 processing cores (with 8×4 generation streams). We use CACTI7 [7] for analysis of power and area for both memory elements. The dynamic memory access is estimated conservatively from simulation trace. The total energy for the whole event queue memory is $\sim$9 Watts when modeled in a 22nm ITRS-HP SRAM logic. Although we use identical systems, GraphPulse accesses 60% less memory than Graphicionado; we did not include DRAM power.

The event collection network is a 16×16 crossbar attached to a network of arbiters allowing groups of Generation Streams to share a port. We modeled a complete RTL design containing the communication network, coalescer engine, and event processors using Chisel and synthesized the model. We assumed that the coalescing pipeline and event processors require floating point units, which results in worst case complexity and power consumption estimates (recall that the coalescing logic is application dependent). The area of the circuit stands at $3.5mm^2$ with a 28nm technology (excluding the on chip memory) and comfortably meets the timing constraint for 1GHz clock. Power estimates show that custom computation modules and the communication network consumes less than 60mW. A breakdown of the power consumption of our evaluated design is presented in Table V. GraphPulse is $280\times$ more energy efficient than Ligra due to the low power from the customized processing and faster overall execution time.

TABLE V
POWER AND AREA OF THE ACCELERATOR COMPONENTS

	#	Power(mW)			Area(mm^2)
		Static	Dynamic	Total	
Queue	64	116	22.2	8825	190
Scratchpad	8	0.35	1.1	11.6	0.21
Network		51.3	3.4	54.7	3.10
Processing Logic		-	-	1.30	0.44

VII. Related Work

Graph Accelerators: Template based graph accelerators process hundreds of vertices in parallel to mask long memory latency [5], [39]. They use hardware primitives for synchronization and hazard avoidance. Swarm [23] allows speculative execution to increase parallelism; however, memory inefficiencies persist. Spatial Hints [24] uses application-level knowledge to tag tasks with specific identifiers for mapping them to processing elements which allows better locality and more efficient serialization, thus, addressing the inefficiencies of Swarm. On the other hand, serialization becomes unnecessary after coalescing in GraphPulse since transaction safety is implicitly guaranteed by the execution model and architecture.

Graphicionado [18], a pipelined architecture, optimizes vertex-centric graph models using a fast temporary memory space. It improves locality using on-chip shadow memory for vertex property updates. However, GraphPulse substantially outperforms Graphicionado due to advantages of event-driven model over conventional models.

PIM-based solutions: These solutions lower memory access latency and increase performance. Tesseract [2] implements simplified general purpose processing cores in the logic layer of a 3D stacked memory. GraphPIM [36] replaces atomic operations in the processor with atomic operation capability in Hybrid Memory Cube (HMC) 2.0 to achieve low latency execution of atomic operations. Our approach has potential to be advantageous on PIM platforms too because memory accesses are simplified and the complex scheduling and synchronization tasks are isolated to the logic layer.

Tolerating irregular memory accesses: The propagation blocking technique for PageRank [8] temporarily holds contributions in hashed bins in memory, merges contributions to same vertex, and later replays them to maximize bandwidth utilization and cache reuse. However, it entails overhead of maintaining bins. Zhou et al. [65], [66] store contributions temporarily to memory and combines them using hardware

when possible for edge-centric model. Due to large number of edges there is a substantial increase in random memory writes to temporary bins and small combination windows limit combining. To optimize irregular memory accesses, IMP [59] uses a dynamic predictor for indirect accesses to drive prefetching. HATS [34] proposes a hardware assisted traversal scheduler for locality-aware scheduling.

Dataflow architectures: GraphPulse bears some similarities to dataflow architectures in that computation flows with the data [12], [21], [26], [53]. The Horizon architecture supports light weight context switching among massive number of threads for tolerating memory latency. It heavily relies on the compiler [12] but dynamic parallelism in graph applications is not amenable to similar compiler techniques. The SAM machine [21] employs a highspeed memory (register-cache) between memory and execution units is a better match for graph applications. However, neither of these architectures address issues in graph processing addressed by GraphPulse. Specifically, GraphPulse uses data carrying events to eliminate random and wasteful memory accesses, coalescing strategy eliminates atomic operations and reduces storage for events, and event queues improve locality and enable prefetching.

VIII. Concluding Remarks

In this paper, we presented GraphPulse, an event-based asynchronous graph processing accelerator. We showed how the event abstraction naturally expresses asynchronous graph computations, optimizes memory access patterns, simplifies computation scheduling and tracking, and eliminates synchronization or atomic operations. GraphPulse achieves an average of 28× improvement in performance over Ligra running on a 12 core CPU implementation, and an average of 6.2× performance improvement over Graphicionado [18].

Acknowledgements

The authors would like to thank the anonymous shepherd and additional reviewers for their detailed feedback and help in improving the paper. This work is supported by grants CCF-2028714, CCF-2002554 and CCF-1813173 from the National Science Foundation to the University of California Riverside and award No. FA9550-15-1-0384 from the Air Force Office of Scientific Research (AFOSR).

References

[1] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1247–1262.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 105–117, Jun. 2015.

[3] Amazon AWS. Amazon EC2 F1 Instances. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[4] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, vol. 11, 2011.

[5] A. Ayupov, S. Yesil, M. M. Ozdal, T. Kim, S. Burns, and O. Ozturk, "A template-based design methodology for graph-parallel hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 420–430, Feb 2018.

[6] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 44–54.

[7] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 14:1–14:25, Jun. 2017.

[8] S. Beamer, K. Asanović, and D. Patterson, "Reducing Pagerank communication via propagation blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 820–831.

[9] E. Bullmore and O.Sporns, "Complex brain networks: graph theoretical analysis of structural and functional systems," in *Nature Reviews Neuroscience, 10(3)*, 2009, pp. 186–198.

[10] P. Burnap, O. F. Rana, N. Avis, M. Williams, W. Housley, A. Edwards, J. Morgan, , and L. Sloan, "Detecting tension in online communities with computational Twitter analysis," in *Technological Forecasting and Social Change, 95*, 2015, pp. 96–108.

[11] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun, "Hardware acceleration of transactional memory on commodity systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 27–38.

[12] J. M. Daper, "Compiling on Horizon," in *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '88, 1988, pp. 51–52.

[13] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[14] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 293–304.

[15] M. D. Domenico, A. Lima, P. Mougel, and M. Musolesi, "The anatomy of a scientific rumor," *Scientific Reports*, vol. 3, 2013.

[16] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, M. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *USENIX OSDI*, 2014, pp. 599–613.

[17] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[18] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[19] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010, pp. 37–47.

[20] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *PVLDB*, vol. 8, no. 9, pp. 950–961, 2015.

[21] H. Hum and G. Gao, "Efficient support of concurrent threads in a hybrid dataflow/von Neumann architecture," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, ser. IEEE IPDPS '91, 1991, pp. 190–193.

[22] H. Isah, P. Trundle, , and D. Neagu, "Social media analysis for product safety using text mining and sentiment analysis," in *14th UK Workshop on Computational Intelligence (UKCI)*, 2014.

[23] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 228–241.

[24] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-centric execution of speculative parallel programs," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.

[25] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[26] J. T. Kuehn and B. J. Smith, "The horizon supercomputing system: architecture and software," in *Supercomputing '88:Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I*, 1988, pp. 28–34.

[27] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 591–600.

[28] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi : Large-scale graph computation on just a PC," in *USENIX OSDI*, 2012, pp. 31–46.

[29] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[30] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[31] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[32] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 25:1–25:39, Oct. 2015.

[33] J. Model and M. C. Herbordt, "Discrete event simulation of molecular dynamics with configurable logic*," in *2007 International Conference on Field Programmable Logic and Applications*, 2007, pp. 151–158.

[34] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Oct 2018, p. 1–14.

[35] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1009–1022.

[36] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 457–468.

[37] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *ACM SOSP*, 2013.

[38] D. Omand, J. Bartlett, and C. Miller, "Introducing social media intelligence (SOCMINT)," in *Intelligence and National Security, 27.6*, 2012, pp. 801–823.

[39] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 166–177.

[40] D. Petrović, T. Ropars, and A. Schiper, "Leveraging hardware message passing for efficient thread synchronization," *ACM Trans. Parallel Comput.*, vol. 2, no. 4, Jan. 2016.

[41] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, no. 6, p. 12–25, Jun. 2011.

[42] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 13–24, Jun. 2014.

[43] S. Rahman, N. Abu-Ghazaleh, and W. Najjar, "PDES-A: a parallel discrete event simulation accelerator for FPGAs," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2017, pp. 133–144.

[44] S. Rahman, N. Abu-Ghazaleh, and W. Najjar, "PDES-A: Accelerators for parallel discrete event simulation implemented on FPGAs," *ACM Trans. Model. Comput. Simul.*, vol. 29, no. 2, Apr. 2019.

[45] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 37–42, Mar. 2011.

[46] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[47] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[48] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 445–456.

[49] J. Shun and G. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 135–146.

[50] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, Oct 2014, p. 481–490.

[51] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2017, p. 646–655.

[52] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *Proceedings of the International Scientific Conference and International Workshop Present Day Trends of Innovations*, 2012.

[53] M. R. Thistle and B. J. Smith, "A processor architecture for Horizon," in *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '88, 1988, pp. 35–41.

[54] A. L. Traud, P. J. Mucha, and M. A. Porter, "Social structure of Facebook networks," *Phys. A*, vol. 391, no. 16, pp. 4165–4180, Aug 2012.

[55] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[56] K. Vora, S. C. Koduru, and R. Gupta, "Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 861–878.

[57] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic i/o optimization for disk-based graph processing," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 507–522.

[58] Yahoo! Research. YAHOO! Webscope Program. [Online]. Available: http://webscope.sandbox.yahoo.com/

[59] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.

[60] G. Zhang, W. Horn, and D. Sanchez, "Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, p. 13–25, citation Key: COUP.

[61] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *CoRR*, vol. abs/1710.05785, 2017.

[62] Y. Zhang, X. Liao, H. Jin, L. Gu, and B. B. Zhou, "FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 5, pp. 895–907, 2018.

[63] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing ordered graph algorithms with GraphIt," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 158–170.

[64] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 121:1–121:30, Oct. 2018.

[65] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on FPGA," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 103–110.

[66] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An FPGA framework for edge-centric graph processing," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18. New York, NY, USA: ACM, 2018, pp. 69–77.