# PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators

Vidushi Dadu Sihao Liu Tony Nowatzki
University of California, Los Angeles
{vidushi.dadu,sihao,tjn}@cs.ucla.edu

*Because of the importance of graph workloads and the limitations of CPUs/GPUs, many graph processing accelerators have been proposed. The basic approach of prior accelerators is to focus on a single graph algorithm variant (eg. bulk-synchronous + slicing). While helpful for specialization, this leaves performance potential from flexibility on the table and also complicates understanding the relationship between graph types, workloads, algorithms, and specialization.*

*In this work, we explore the value of flexibility in graph processing accelerators. First, we identify a taxonomy of key algorithm variants. Then we develop a template architecture (PolyGraph) that is flexible across these variants while being able to modularly integrate specialization features for each.*

*Overall we find that flexibility in graph acceleration is critical. If only one variant can be supported, asynchronous-updates/priority-vertex-scheduling/graph-slicing is the best design, achieving 1.93× speedup over the best-performing accelerator, GraphPulse. However, static flexibility per-workload can further improve performance by 2.71×. With dynamic flexibility per-phase, performance further improves by up to 50%.*

## I. Introduction

Graphs are fundamental data structures in data mining, navigation, social networks and AI. Graph processing workloads are challenging for traditional architectures (CPUs/GPUs) due to data-dependent memory access, reuse, and parallelism. However, these workloads present many specialization opportunities: commutative updates, a resilience to performing work in an arbitrary order, and repetitive structure in memory access and computation. This implies large advantages for specialization, either with CPU offload engines [18,19,28,29,52] or accelerators [1,2,12,16,32,49,50,54,61].

Yet these designs generally make strong assumptions about certain aspects of graph processing for simplicity, which can limit their ability to generalize. This includes assuming a certain input graph type (eg. high vs. low diameter), workload property (eg. order resilience, frontier density), and most importantly for this work: fundamental *graph algorithm variants*.

These variants dramatically affect the tradeoff between throughput and work-efficiency[1]: 1. Update Visibility: granularity when graph updates become visible, 2. Vertex Scheduling: the fine-grain scheduling policy for vertex updates, 3. Slice Scheduling: whether and how the graph working set is controlled, and 4. Update Direction: Whether vertices update their own or neighbors' properties (pull/push). Each variant has profoundly different implications on hardware, and these implications can vary by graph and workload type.

[1]Work-efficiency is the work-required by the optimized sequential execution (in terms of edges processed) over the work performed in parallel execution.

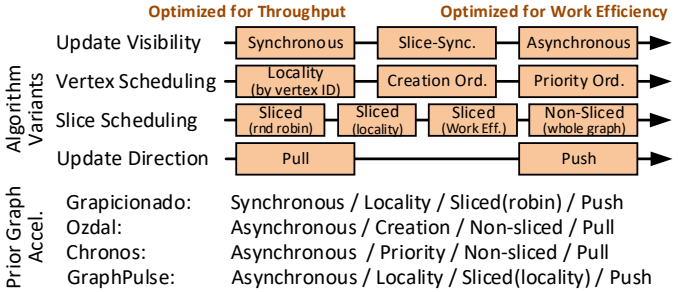



**Fig. 1: Algorithm Variant Dimensions & Prior Accelerators**

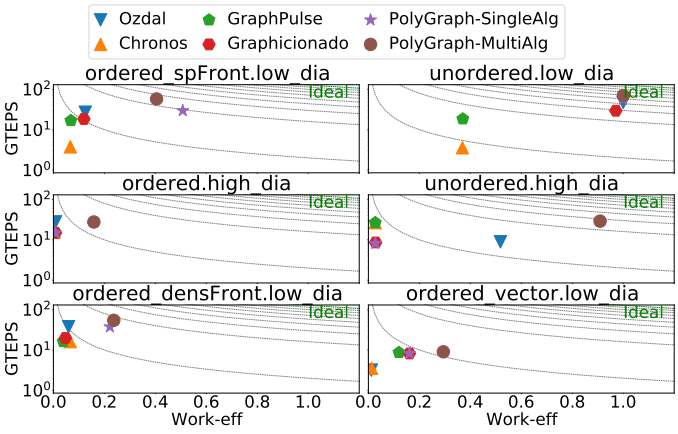



**Fig. 2: Work-efficiency and Throughput Tradeoffs**

Most prior graph-accelerators have each focused on one *algorithm variant* (see Figure 1). Because each algorithm variant causes different tradeoffs in work-efficiency, locality/memory-efficiency, and load balance, different accelerators end up performing well for different workloads and input graphs. Figure 2 shows the throughput (in giga-traversed edges per second - GTEPS) versus work-efficiency measured by our model of these accelerators. Performance is a product of work-efficiency and throughput, so we also show equi-performance curves in this figure. Our insight is that we can optimize performance by having the flexibility to use the right algorithm variant for the right graph and workload (or even workload phase).

Selecting the right variant is simple; the challenge is to design an architecture with sufficient algorithm/architecture flexibility, and little performance, area, and power overhead. This flexibility requires supporting different granularity tasks (synchronous vs asynchronous updates), fine-grain task scheduling, flexibly controlling the working set, and having flexibility for different data structures.

**Approach:** Our design augments prior efficient decoupled-spatial accelerators [11,23,31,34,55], which can support general data-structures and suit both memory-intensive (eg. Breadth-first Search (BFS)) and compute-intensive workloads (eg. Graph Convolutional Networks (GCN)). The fundamental limitation of existing decoupled-spatial accelerators is the lack of support for fine-grain data-dependent parallelism (ie. task parallelism). To solve this, we developed a novel execution model, called Taskflow, that integrates task abstractions and data-dependent scheduling as first-order primitives within a dataflow program representation.

Integrating this flexible hardware and variant combinations required solving several challenges. First, integrating asynchronous variants with sliced execution required new mechanisms for deciding when to switch slices and how to orchestrate data during slice transition. Next, because tasks can be short lived due to pipelined dataflow execution, we needed to develop a high-throughput task scheduler. Also, because we relied on a mesh interconnect (for scalability and high local bandwidth), a new multi-level spatial partitioning scheme was critical to ensure locality and balanced load.

Our accelerator – PolyGraph – combines the above mechanisms in a programmable multicore specialized for many variants of graph processing, and is well-suited for other workloads with dynamic parallelism and large working sets.

**Evaluation and Results:** We evaluated PolyGraph with cycle-level simulation, supporting a design space of architectures with features encompassing many prior works [1,32,50]. We evaluated traditional and ML-based graph workloads (Table I), on graphs with up to 1.5 billion edges. The best algorithm-specific PolyGraph design (PG-singleAlg) is 16.79× (up to 275× for high diameter graphs) faster than a Titan V GPU. More importantly, we find that flexibility is critical. By statically choosing the best algorithm variant, we gain 2.71× speedup. Dynamic flexibility provides 1.09× further speedup.

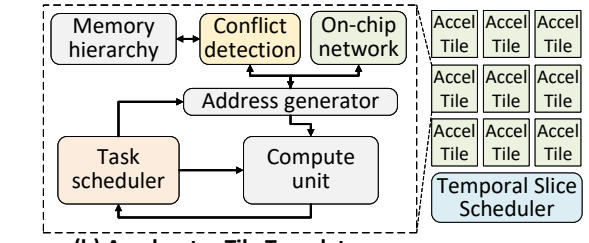
**Contributions:**

- Idea of flexibly supporting and dynamically switching between specialized graph processing algorithm variants.
- Flexible "taskflow" execution model, integrating dynamic task parallelism and pipelined dataflow execution.
- Novel $\mu$architectural support for pipelined task creation and scheduling, leveraging properties of graph algorithms.
- Broad and detailed comparison of prior graph accelerators.
- Insights into the relationships between graph types, workloads, algorithm variants, and architecture techniques.

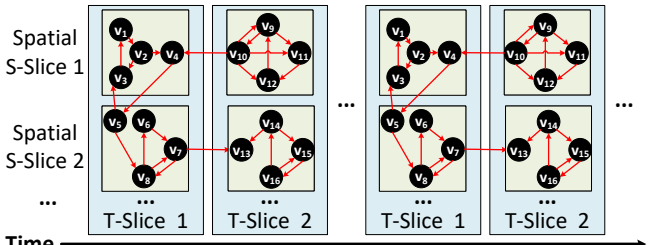
**Paper Org.:** We first give background on key workload/graph properties (§II). We then discuss a taxonomy of algorithm variants (§III). We develop a unified program IR for these variants (§IV), and describe the support within PolyGraph (§V), as well as our novel spatial partitioning (§VI). Finally, we evaluate and discuss related work (§VII,VIII,IX).

```
1: procedure CALC_SHORTEST_PATH(GRAPH=(V,E,
   T:Temporal Slices (optional); S:Spatial Slices))
2: vert_prop[V_init] = 0 # initialize vertex prop
3: active_set[T][S] = {} # initialize active sets
4: add V_init to active_set
5: while any active slice left
6:   Choose temporal slice t
7:   for all spatial slices s in t: do in parallel
8:     while any active vertex active_set[t][s]
9:       Chose vertex v
10:      for all outgoing vertices w: do
11:          new_prop = vert_prop[v]+edge_wgt[v->w]
13:          if vert_prop[w]-new_prop[w] > 0 then
14:            vert_prop[w] = new_prop[w]
15:            add w to active_set
```

(a) SSSP Algorithm (temporally sliced algorithm variant)




(b) Accelerator Tile Template




(c) Logical execution of slices over time

**Fig. 3: Algorithm (SSSP) and Mapping to Arch. Template**

## II. Graph Acceleration Background

Here we describe our computational paradigm, mapping to a template architecture, and key graph/workload properties.

### A. Vertex-centric, Sliced Graph Execution Model

In vertex-centric graph execution [17,25,26,38,42,48], a user-defined function is executed over vertices. This function accesses properties from adjacent vertices and/or edges, and execution continues until these properties have converged.

Preprocessing the graph can offer better spatial and/or temporal locality. Commonly, the graph is divided into *temporal slices* (or T-slices) which fit into on-chip memory. Further preprocessing may divide the graph among cores for load-balance or locality; these are *spatial slices* (S-slices).

**Example: Shortest Path (SSSP):** Figure 3(a) shows an example code (SSSP) written in this model. An active list is maintained for each temporal and spatial slice. After initialization,

| | Priority | Vertex comp. | Vertex update | Prop(B) |
|---|---|---|---|---|
| SSSP [39] | dist | src_dist+edge_wgt | min(dist,dst_dist) | 4B |
| BFS [39] | depth | src_depth+1 | min(depth,dst_depth) | 4B |
| CC [39] | comp. ID | - | min(src_id,dst_id) | 4B |
| PR [47] | res/deg | src_res*$\alpha$/src_deg | new_res+dst_res | 2x4B |
| CF [58] | grad | f(src_prop.dst_prop) | new_vec+dst_vec | 32x4B |
| GCN-Inf [22] | - | matrix-mult | src_vec+dst_vec | 128x4B |

**TABLE I: Graph Workloads** (Prop: vertex prop. size).

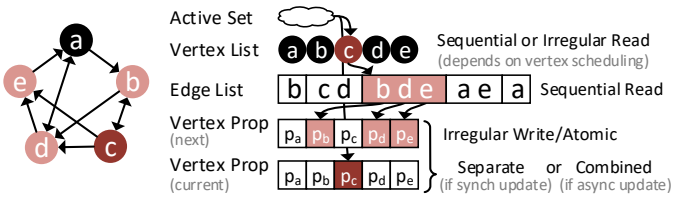



**Fig. 4: Graph Data Structures**

the algorithm iterates over all active temporal slices until no vertex is active. Within each temporal slice, the corresponding spatial slices execute concurrently (see Figure 3(c)). Within each spatial slice, the vertices are scheduled iteratively. Each vertex execution updates its neighbor's vertex properties. If the destination's vertex property is changed, it is activated[2].

Different graph workloads can be implemented by changing: 1. initial active vertices, 2. the function computed for each vertex, 3. the update function for the destination vertex. An optional characteristic is a priority hint for vertex scheduling (see Figure 3(c), Line 11). Examples are shown in Table I.

Figure 3(b) shows a high-level decoupled-spatial template architecture, colored to indicate the relationship between algorithm and accelerator. The temporal-slice scheduler chooses a T-slice in each coarse graph phase. The spatial slices (S-slices) are executed in parallel across all cores. A task scheduler picks a vertex to execute at each step, and the per-vertex computation (lines 10-15) is mapped to the address generator and compute unit. Atomic updates to vertex properties (lines 13-14) are enforced using conflict detection and stalling.

**Graph Data-structures:** Figure 4 overviews the essential data structures. A vertex-list stores the first edge index for each vertex. Edges are stored contiguously in the edge list, containing a destination vertex_id (and optional edge weight). Each vertex has a property which the algorithm computes. The active list data structure, and whether we double buffer the vertex properties, depends on the algorithm variant (§III).

### *B. Key Workload/Graph Properties*

**Graph Property: Diameter:** is the largest distance between two vertices. *Uniform-degree graphs* (eg. roads) have a similar/low number of edges-per-vertex, and thus a high diameter. Oppositely, *power-law graphs* (eg. social networks) have low diameter, as some vertices are highly connected.

**Workload Property: Order Sensitivity:** Many graph workloads are *iterative* and *converging*. This converging nature tends to make such workloads functionally resilient to the order in which computation occurs. However, some of these

[2]This describes the *push*-based update direction. For *pull*, the incoming edges are used to update the vertex's own property.

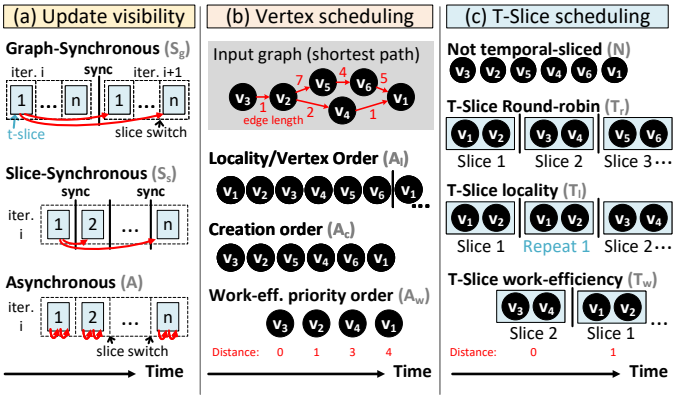



**Fig. 5: Key Variants of Graph Processing Algorithms**

workloads may require a different amount of work depending on the order tasks are performed; ie. their work-efficiency is *order-sensitive*. For example, in a shortest path algorithm, exploring a farther away vertex before a near vertex can lead to redundant work, because the distance of the farther vertices may be updated if a shorter path is found. Sensitivity varies with workload. Breadth-first search (BFS), is less sensitive as many paths are of equal depth, making it less likely to find a wrong path. Non-converging workload like GCN is order insensitive.

**Workload Property: Frontier Density:** Dense frontier workloads like PR, CF usually have more than 50% active vertices while sparse frontier workloads (eg. SSSP, BFS) require much fewer. In general, sparse frontier workloads require fewer passes through the graph until convergence.

## III. Graph Algorithm Taxonomy

To systematically study the value of flexibility, we create a taxonomy of four key dimensions and discuss tradeoffs.

**Update Visibility:** defines when writes become visible to other computations, and hence this affects the granularity at which new tasks are created. Writes may become visible after one pass through the graph (*graph-synchronous*), after each slice (*slice-synchronous*)[3], or immediately (*asynchronous*). Barriers are used to synchronize update propagation in synchronous variants. Figure 5(a) visualizes dependences in a slice-based execution of graph processing (blue boxes are graph slices which fit in on-chip memory). The figure shows how the dependence distance shrinks (red arrows) moving from synchronous to asynchronous.

*Tradeoffs:* While synchronous algorithms naturally provide sequential consistency of vertex updates, fast implementations of asynchronous variants do not lock neighboring vertices, and hence do not provide sequential consistency of tasks. While many converging workloads do not require this, some workloads may not be expressible or converge slower (eg. ALS [25]). Also, barriers in synchronous variants can be a high overhead when the work between them is low (eg. due to few active vertices).

[3]Graph/slice synchronous are bulk-synchronous [41] at different granularity.

**Vertex Scheduling:** defines the processing order for active vertices, relevant for asynchronous variants. Figure 5(b) depicts the variants for shortest path: *Locality order:* To improve the vertex access locality, schedule by vertex-id. *Creation order:* Schedule vertices in the order they are activated (breadth-first in the figure). *Work-efficiency order:* Schedule vertices in an order which reduces redundant work (by distance in the figure). Section II-B explains the intuition. Table I lists the priority metric for each workload.

*Tradeoffs:* When active vertices are accessed in their storage order, spatial locality enables high memory bandwidth; However, this costs work-efficiency, as it requires critical updates to be delayed. Creation order requires simple FIFO logic, while Work-efficiency order requires dynamic sorting. Note that distributed ordering is sufficient [47].

**Temporal Slicing:** defines whether the working set may be limited to a predefined *slice* of all graph vertices. Slices are determined during offline partitioning and are generally sized to fit on-chip memory. Updates to data outside the current slice are deferred, and an explicit phase is required to switch slices (combined with barriers in synchronous variant). Slices can be scheduled in different orders, forming new variants. Figure 5(c) depicts each: *Round-robin:* iterate through all slices. *Locality:* similar, but repeatedly process each slice. *Work-efficiency:* prioritize slices whose properties change most [51]. In the example, slice 1 is chosen second, as its properties changed most (v1 and v2).

*Tradeoffs:* Non-sliced avoids barriers and slice-switching data movement, which is costly if there are few active vertices. Slicing can also harm the optimal ordering by restricting the scheduling scope. The key benefit of temporal-slicing is more effective on-chip memory use. Locality scheduling improves intra-slice reuse but may delay cross-slice updates. Sliced-work-efficiency ordering optimizes for work-efficiency without requiring hardware support for fine-grained scheduling.

**Update Direction:** defines whether a task updates its own property (pull/remote read), or whether a task updates its neighbor's properties (push/remote atomic update).

*Tradeoffs:* Push reduces communication bandwidth by using one-way communications (push updates to neighbors) and efficient multicast [5], rather than the remote memory requests in pull. The request latency is hidden due to the access reordering potential of push. Finally, pull often requires more work while reading all incoming edges of each active vertex. However, there are techniques that optimize pull by eliminating edge accesses based on vertex convergence [3,60]: their effectiveness depends on prefetching capability.

**Notation:** We use a two-letter shorthand (sometimes expanded) to denote each variant combination (Figure 6). By default we assume push; we notate pull with an explicit suffix.

**Summary:** Table II summarizes the throughput (lacks synchronization or improves locality) or work-efficiency benefit (reduces update latency or has priority ordering); it also shows which graph type and #active-vertices it is best suited for.

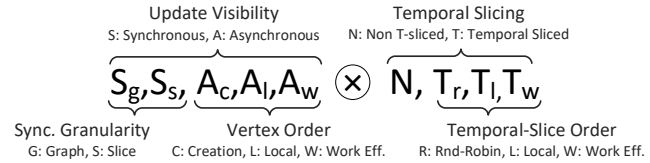



**Fig. 6: Shorthand for Algorithm Variants**

| | | Throughput benefit: No Sync | Throughput benefit: Mem b/w | Throughput benefit: Latency | Work-eff. benefit: Prio. | Inputs optimized for: Order Sens. | Inputs optimized for: Graph | Inputs optimized for: Active Tasks |
|---|---|---|---|---|---|---|---|---|
| Update-Vis. | $S_g$ | | ✓ | | | | Low dia | More active |
| | $S_s$ | | ✓ | ✓ | | ✓ | Low dia | More active |
| | $A_c$ | ✓ | | ✓ | | ✓ | High dia | Less active |
| | $A_l$ | ✓ | ✓ | ✓ | | | High dia | Less active |
| | $A_w$ | ✓ | | ✓ | ✓ | ✓ | High dia | Less active |
| T-Slicing | $N$ | ✓ | | ✓ | | ✓ | High dia | Less active |
| | $T_r$ | | ✓ | | | | Low dia | More active |
| | $T_l$ | | ✓ | | | | Low dia | More active |
| | $T_w$ | | ✓ | | ✓ | ✓ | Low dia | More active |

**TABLE II: Algorithm Variant Tradeoffs**

## IV. Unified Graph Processing Representation

To support variants efficiently on a unified hardware, we need a program representation that is flexible, fast, and is specialized for graph workloads. Because of their different needs, we develop separate approaches for the data-plane (pipelined task execution) and the control plane (slice scheduling).

### A. Data plane Representation: Taskflow

Three major requirements drive taskflow's design: 1. Need for fully pipelined execution of per-vertex computation. 2. Need to support data-dependent creation of new tasks, including programmatically specifying and updating the priority ordering. 3. Need for streaming/memory reuse.

*Taskflow* satisfies these requirements by augmenting a dataflow model with first-class support for priority ordered tasks. In taskflow, a task is invoked by its type `t` and input arguments: `<t,args>`. We use a type rather than a function, because graph workloads typically only require 2 or 3 tasks types, and this makes supporting reconfigurable hardware straightforward. Each task type is defined by a graph of compute, memory and task nodes:

- **Compute nodes:** are passive, and may maintain a single state item. This enables them to be mapped to systolic-like fabrics [9,11,15,31,34] for high efficiency.
- **Memory nodes:** represent decoupled patterns of memory access, called streams [7,11,21,31,44,45]. Stream parameters can either be constant (set at stream creation) or dynamic (consumed from another node with a FIFO interface). Figure 7(a) defines stream parameters and behaviors.
- **Task nodes:** represent arguments, and are ingress and egress points of the graph. An instance of a task is started by providing a value to each of the ingress task nodes, and a task is created when values arrive at all egress task nodes.

**Atomics:** Shared-memory atomics are critical due to the need for correct handling of memory conflicts on vertex updates. In

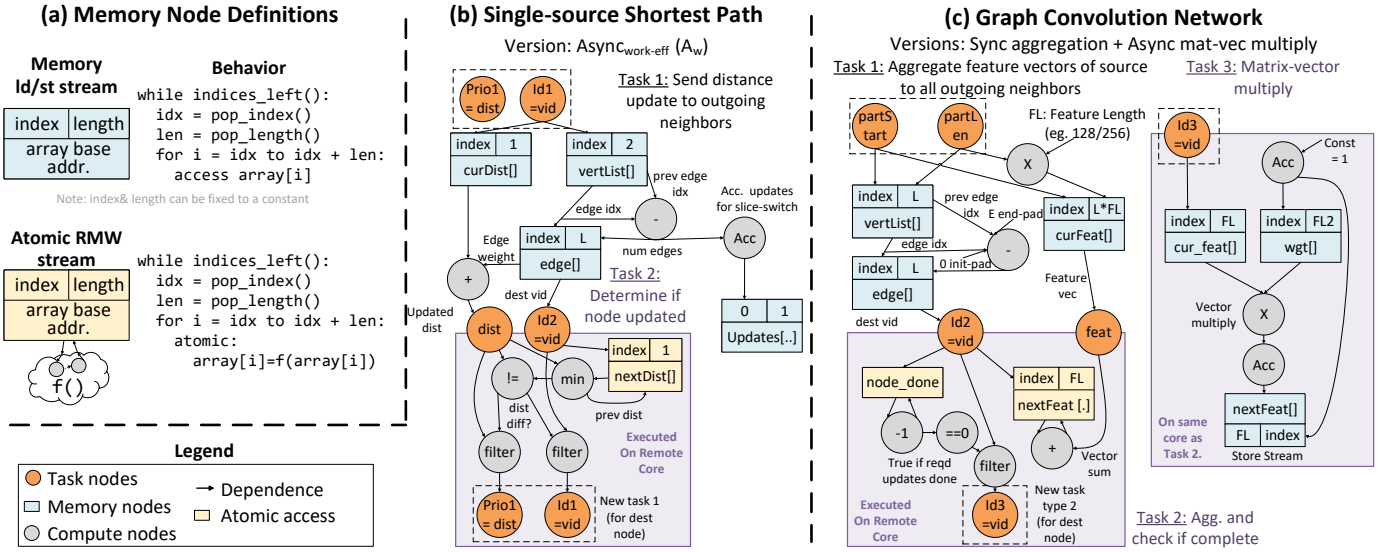



**Fig. 7: Taskflow Examples**

taskflow, a memory stream can be marked as "read-modify-write" (RMW) and will be atomic. See example in Figure 7(a).

**Priority Scheduling and Coalescing:** One task argument may be designated at compile time as the task's priority. The parent task computes the priority prior to task creation, and this serves as a hint to schedule tasks with higher priority first.

Another task argument indicates the ID, which is unique for all active tasks. The ID generally is the vertex_id, and may be used for task coalescing and sliced execution. When two tasks with the same ID are created, this is treated as an update to the priority of the original task, and the two tasks are "coalesced" into one higher priority task. The upper bits of the ID indicate the task's T-slice and S-slice. Tasks are deferred until their T-slice is active and are scheduled at their S-slice core.

**Taskflow Examples:** Figure 7(b) shows an example taskflow graph for SSSP, implemented as $\text{Async}_{work-eff}$. This workload has two tasks, each associated with different vertex data: Task type 1 iterates over outgoing edges of a source vertex to compute distances, and creates a type 2 task for each destination vertex to carry out distance updates. The ID of task 1 and 2 is the vertex-ID of source and destination vertices respectively, and they execute on the core corresponding to the ID. Type 1 tasks are prioritized by vertex distance for work-efficiency. Type 2 tasks also check if the vertex should become active, and if so, a new type 1 task will be created. The number of task 1 invocations is accumulated in a shared memory location to identify slice-switching (see §IV-B).

In GCN, graph aggregation is synchronous, while matrix-vector multiply is asynchronous – this enables overlapping the memory-intense aggregation tasks with compute-intensive matrix multiply. Here, task 1 is *coarse-grained*, and iterates over all vertices (in a slice), and a single task node trigger creates many cycles of work. It pushes source vector features to task 2 to aggregate them together for incoming edges of each vertex. Aggregation (task 2) asynchronously triggers a matrix-vector multiply (task 3) when aggregation is complete (identified using node_done). Note that this overlap splits matrix-matrix into multiple matrix-vector computations: this prevents the broadcast of weight matrix. To localize the traffic for weight responses, we duplicate weight matrix at each core (this is low overhead: only consumes 3% of scratch space).

**Taskflow Flexibility Summary:** Synchronous variants ($S_{graph}$,$S_{slice}$) use coarse grain tasks that pass through the (per graph/per slice) active list. Asynchrony is supported with explicit fine-grain tasks, optionally with priority hint argument.

### *B. Slice Scheduling Interface and Operation*

The responsibility of the temporal slice scheduler is to configure on-chip memory, decide which slice to execute next, and manage data/task orchestration. The slice scheduler is invoked infrequently, and can be executed on a simple control core with limited extensions for data pinning operations. The slice scheduler also has a mechanism for creating initial tasks.

**Data Pinning:** Depending on the algorithm variant, we may know which data we want to most reuse. For eg. for synchronous non-sliced ($S_gN$), edges have large reuse distance, but vertices with high-degree are reused many times. In sliced-locality variants, edges are also reused, to a lesser extent. To help the slice-scheduler optimize for reuse, we provide the slice scheduler an interface to *pin* a range of data to the on-chip memory at a particular offset, essentially reserving a portion of the cache (eg. pin the region of vertex properties that have high reuse). Non-pinned data is treated like normal cache access.

**Slice Switching for Asynchronous Variants:** The decision of when to switch slices for asynchronous variants is a tradeoff between work-efficiency (switch sooner) and reuse (switch later). To explain, information at the slice's boundary becomes "stale" over time, as it may depend on an inactive slice's execution, thus hurting work-efficiency. It is impossible to calculate "staleness", as it depends on the future execution. Therefore, we approximate it by counting the number of vertex updates, and switch slices when these exceed a threshold.

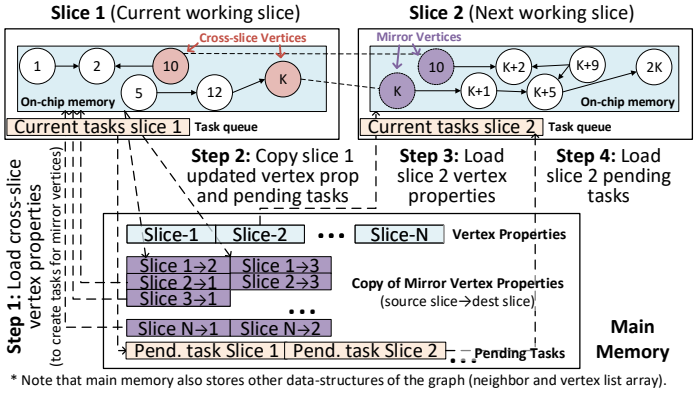



**Fig. 8: T-Slicing for Large Graphs** (N slices, K vertices each)

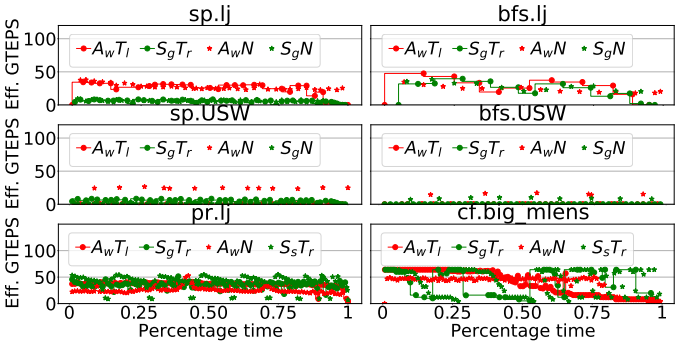



**Fig. 9: Potential of Dynamically Switching Variants** (Effective GTEPS is the useful throughput – "work-done-per-second"/"work-efficiency". Here the work-efficiency is normalized to $A_wN$ and thus, the area under the curve is "$A_wN$-work"/"total-execution time".)

When switching slices, the slice scheduler gives all cores a highest priority stop task, which disallows any new tasks to be issued and it may also perform the slice transition as explained below.

**Slice Preprocessing:** Slices are preprocessed to keep all edges (and hence updates) within each slice. This is accomplished by creating a mirror vertex for any cross-slice edge in the destination slice, and removing the original cross-slice edge from the source slice. For example, if edge A→B crosses a slice, then a mirror vertex for A would be created in B's slice. Mirror A retains the edge A→B, while this is removed in the original vertex A (in A's slice). The mirror vertices properties are only updated during slice transition, as explained next.

**Slice Transition:** Figure 8 shows how data is orchestrated during slice transition. Two slices are shown, and only one may reside in on-chip memory at a time. Main memory contains the graph vertex properties, pending tasks for each slice, and a copy of each mirror vertex. Transition works as follows:

*Step 1.* Stream in cross-slice vertex properties (eg. vertex 10, K), and scatter to their S-slice core; compare old and new properties to see which vertices changed. If such a property changed, a new task is created in the destination slice by pushing the task's arguments to that slice's pending tasks list (10, K to slice 2).

*Step 2.* Meanwhile, the updated cross-slice vertex properties are stored in the copy of mirror vertex properties. Also, the current slice's pending tasks and updated vertex properties are streamed to memory.

*Step 3.* In parallel with step 2, using double buffering, vertex properties in the next slice are streamed to pinned memory.

*Step 4.* Stream next slice's pending tasks from main memory.

### C. Scheduling of Algorithm Variants

**Quantitative Motivation:** Figure 9 shows the fine-grained effective throughput (normalized to work-efficiency) over time for several algorithm variants. Time on the x-axis is normalized to the percentage of total execution time, as time-scales vary significantly between variants. Notice that the highest performance variant changes during the execution:

*Low Diameter Graph (lj):* For order-sensitive SSSP (`SP` in figures), $\text{Async}_{\text{work-eff}}$ dominates due to work-efficiency gains,

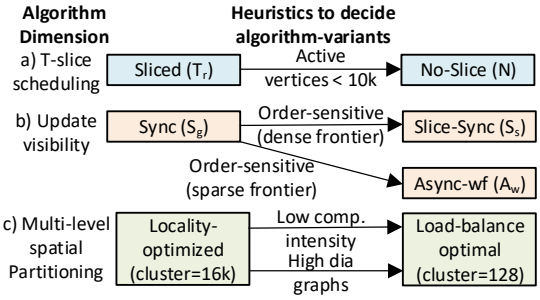



**Fig. 10: Algorithm Variant Scheduling**

while for less order-sensitive BFS, both $\text{Async}_{\text{work-eff}}$ and $\text{Sync}_{\text{graph}}$ are similar. The sliced versions improve on-chip hit rate, however switching to non-sliced at beginning/ending iterations has significant potential.

*High Diameter Graph (USW):* Since high diameter graphs have only a few vertices active in each iteration, synchronization overhead is critical. Therefore, $\text{Async}_{\text{work-eff}}$No-slice consistently dominates.

*Dense frontier workloads (pr,cf):* For PageRank, there is a tension between memory efficiency due to high active vertices and work-efficiency due to order sensitivity. Therefore, $\text{Slice}_{\text{sync}}$ balances tradeoffs and is optimal. For CF, $\text{Async}_{\text{work-eff}}\text{T}_{\text{locality}}$ is optimal in initial iterations, however $\text{Async}_{\text{work-eff}}$No-slice dominates in the later iterations when #active vertices are low.

We found that switching between synchronous/asynchronous hurts work-efficiency, as they proceed differently: asynchronous leaves many vertices active because it focuses on high priority vertices, while synchronous conservatively tries to complete the work for all active vertices in every iteration.

**Heuristics for Algorithm Variant Scheduling:** Figure 10 shows how we decide the algorithm variant. For slicing, the effective throughput depends on whether the work during phase is sufficient to hide barrier overhead. This work can be approximated from active vertices – for example, non-sliced outperforms at the beginning and end of the algorithm, as active vertices are fewer (Figure 10, `lj`). For uniform graph (`rdUSE`), non-sliced consistently outperforms as active vertices are low due to their high diameter. Therefore, our algorithm switches to non-sliced when number of active vertices are below a threshold. In the update visibility dimension,

the decision depends on the workload characteristics. Asynchronous versions are preferred for order-sensitive workloads (See Figure 10, `sp`). Dense frontier algorithms (those with usually >50% active vertices) have high inherent spatial locality, so $\text{Slice}_{\text{sync}}$ is preferred, as it is memory efficient while maintaining moderate work-efficiency. We also propose a flexible multi-level spatial partitioning that can optimize for load and locality depending on the workload and graph type (details explained in §VI). Note that the scheduling decisions may change depending on hardware parameters, as we discuss in Section VIIIE.

**Variant Transition:** Variant selection is performed after a single round of graph/slice in $S_g/S_s$, or after every 100k cycles for asynchronous variants (long enough to amortize the latency of switching). To transition, given the algorithm variant, the control core will: 1. initialize data-structures and configure taskflow graph, and 2. perform pinning operations. If a dynamic switch is invoked, on-chip memories are flushed, and taskflow may require reconfiguration. The pending tasks are managed as during slice transition data orchestration.

## V. PolyGraph Hardware Implementation

PolyGraph is a multicore decoupled-spatial accelerator connected by 2D triple mesh networks[4], overviewed in Figure 11.

The **data plane** is comprised of all modules besides the control core, and is responsible for executing taskflow graphs. The decoupled execution of memory/compute is similar to prior decoupled-spatial accelerators: memory nodes are maintained on stream address generators, and accesses are decoupled to hide memory latency. A Softbrain-like CGRA [31] executes compute nodes in pipelined fashion. Between the stream controller and CGRA are several "ports" or FIFOs, providing latency insensitive communication. The novel aspect is task management: A priority-ordered task queue holds waiting tasks. Task nodes define how incoming task arguments from the queue are consumed by the stream controller to perform memory requests for new tasks.

The basic operation is as follows: If the stream controller can accept a new task, the task queue will issue the highest-priority task. The stream controller will issue memory requests from memory nodes of any active task. The CGRA will pipeline the computation of any compute nodes. The CGRA can also create new tasks by forwarding data to output ports designated for task creation, and these are consumed by the task management hardware. Tasks may be triggered remotely to perform processing near data. Initial tasks may be created by the control core, by explicitly pushing task arguments to the task creation ports.

The **task management unit** enables high-throughput priority-ordered task dispatch. This is critical, as many tasks are short-lived. Therefore, the requests of multiple tasks should be pipelined. The task unit also coalesces superfluous tasks at high throughput (for priority update), by maintaining a

[4]Multiple networks enable efficient scalar remote accesses.

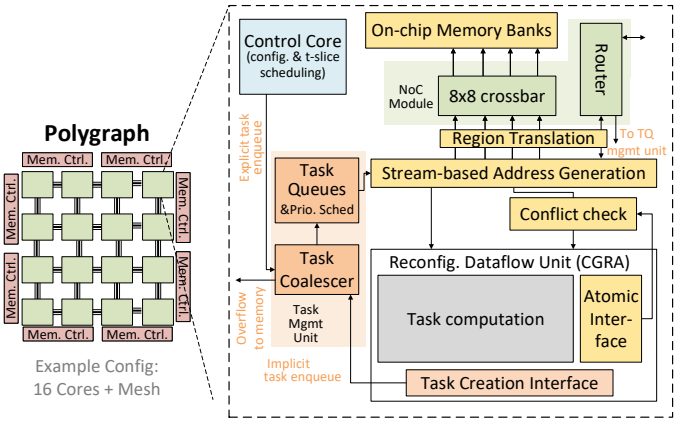



**Fig. 11: PolyGraph Modular Hardware Implementation**

bitvector of in-flight tasks IDs. Tasks can overflow the task queue, which is handled by an overflow protocol.

Finally, **slice scheduling** is implemented on core 0's control core (a simple inorder core). The PolyGraph cores communicate with the slice scheduler through shared memory atomics to coordinate at phase completion. The slice scheduler orchestrates data when switching slices and may initiate a stop task on remote cores to prevent any new tasks to be issued.

### A. Task Hardware Details

**Task Queue and Priority Scheduling:** A task argument buffer maintains the arguments of each task instance before their execution. Statically, it is split into the number of task types and each partition is configured to the size of its corresponding task type arguments. The task argument pointers to ready tasks (ie. whose all arguments are available) are stored in the task scheduler. Note that we use the priority task scheduler (described next) only for graph access tasks and FIFO scheduling for others (eg. vertex update).

Our task scheduler uses a pipelined hardware-based heap [6], where each memory bank represents a level in the priority heap. Push/pop operations move nodes across levels, locking in a hand-over-hand fashion. This implies a throughput of one enqueue-dequeue operation every two cycles. For low degree graphs, this throughput is insufficient. Therefore, we use multiple priority-heaps per core, and alternate between them. This implies imperfect ordering, because two consecutive highest-priority elements could be in the same queue. In practice this does not significantly hurt work efficiency.

**Overflow and Reserved Entries:** If the task queue is full, new tasks will overflow into a buffer in main memory (32kB is sufficient). This buffer is drained to the queue as entries are freed, and the priority then is re-calculated by using the updated *vertex_prop*. Re-calculation is required as the priority might have been updated due to coalescing.

Overflow unfortunately disrupts the priority order, and can hurt work-efficiency due to delaying a high priority task. To mitigate, we reserve some task queue entries for latent high-priority tasks (eg. 32 for 256 sized queue). During overflow,

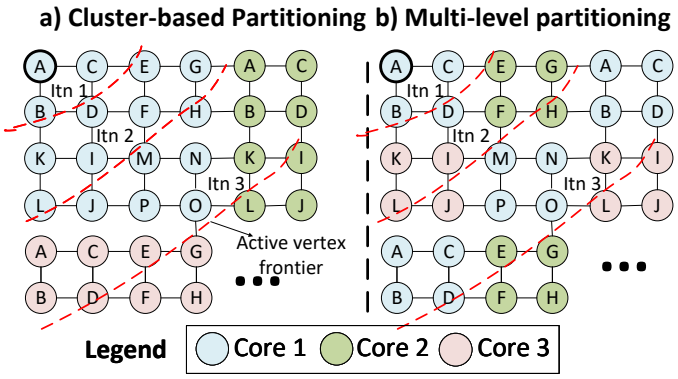



**Fig. 12: Cluster-based vs Novel Multi-level Spatial Partition**

a new task is allocated a reserved entry if it is equal/higher priority than the current highest-priority task.

**Task Coalescing:** To reduce active tasks, we allow coalescing of tasks with the same ID. We implement this with an SRAM bitvector, where each bit corresponds to a task ID. A set bit means the corresponding task is present in the overflow buffer, and this will prevent task insertion for that ID.

For graph processing, task ID corresponds to vertex_id. We size the bitvector to be 32 kB, as this covers the maximum vertices in temporally sliced variants. For non-sliced, PolyGraph only coalesces the first 32K vertex_id. This is sufficient in practice, as the partitioner can simply move critical high-degree vertices to the beginning of the vertex list.

### *B. Memory Architecture*

**Shared Memory:** Our on-chip memory is a shared address-partitioned cache, with multiple banks per PolyGraph core. Each cache bank has a region translation unit that maintains the mapping of virtual address ranges to the pinned addresses.

Data-structure pinning is supported similar to prior buffer-in-NUCA techniques [8]. When the slice-scheduler in core 0 pins a memory region, the region translation unit in all cores are sent the base/bound of the region, along with an offset. This causes some of the sets of the cache to be set aside for pinned data. The core will generate requests for that region and send an acknowledgment when complete. Subsequent memory requests check the range registers to determine if they are to be mapped to the cache or pinned region. Pinning a new data-structure flushes all cache regions in the newly pinned addresses. Re-pinning is only required during transition between variants: this happens at most twice (when active vertices go above or below a threshold), thus the overhead is low compared to the 10s/100s of slice switchings.

**Atomic Updates:** When update requests are received from the local core or the network, they are pushed to the pending atomic requests queue at its corresponding bank. The conflict check logic uses a small CAM (8 entries, to cover atomic latency) to detect and delay aliasing requests.

## VI. Spatial Partitioning

While offline partitioning is common for creating temporal-slices, we find that spatial partitioning makes the mesh-based network highly effective, as we describe next.

| | GPU [10] | Graphcnd. $S_gT_r$ [16] | GraphPls $A_wT_l$ [35] | Ozdal $A_cN$ [32] | Chronos $A_wN$ [1] | PG (ours) |
|---|---|---|---|---|---|---|
| Compute | GP104 | SIMT-16 | ASIC | D-flow | ASIC | CGRA |
| $+Spad | 14.5MB | 32MB | 32MB | 32MB | 32MB | 32MB |
| FP Unit | 5120 | 1024 | 1024 | 1024 | 1024 | 1024 |
| Mem GB/s | 652 | 512 | 512 | 512 | 512 | 512 |
| Net. type | Bus | XBAR | XBAR | XBAR | 3 mesh | |
| Net. radix | - | 128 | 128 | | 5 core/8 mem | |

**TABLE III: Architecture Characteristics of Baselines**

| | Graphs | Vertices | Edges | Dia | Structure | #T-Slices |
|---|---|---|---|---|---|---|
| All | orkut | 3M | 106.3M | 9 | Power-law | 1/2 |
| | LiveJournal | 4.8M | 68.9M | 16 | Power-law | 2/4 |
| | twitter | 41.6M | 1.4B | 9 | Power-law | 5/10 |
| Search | indoChina | 7.4M | 194M | 200 | Random | 2 |
| | rdUSE | 3.5M | 8.7M | 2897 | Uniform | 1 |
| | rdUSW | 6.3M | 15.2M | 10206 | Uniform | 2 |
| cf | big-mlens | 0.2M | 2.5M | 5 | Power-law | 2 |
| | mlens | 82k | 10M | 5 | Power-law | 1 |
| gcn | pubmed | 0.02M | 0.09M | 9 | Power-law | 2 |
| | cora | 2.7k | 10k | 20 | Power-law | 1 |

**TABLE IV: Input Graphs** (Left column is the domain. PR requires double #T-slices; #T-slices for CF/GCN depends on feature size.)

Spatial partitioning introduces a tradeoff between locality and load balance. Naively clustering connected vertices will reduce network traffic, but may hurt load balance, especially for sparse frontier workloads. We explain with a simple grid-graph[5] in Figure 12, but observations apply more broadly. In Figure 12(a), we use clustering-only for creating S-slices; the dotted red lines show the progression of a sparse frontier workload like BFS. What we can observe is that the frontier in several iterations is limited to a single slice (allocated in one core); hence, this strategy has extremely poor load balance.

**Multi-level Scheme:** Figure 12(b) visually shows our proposed "multi-level" slicing scheme that respects both load balance and locality. First, the graph is split into many small clusters of fixed size to preserve locality, then these clusters are distributed equally among cores for balanced load. To implement, we use a simple bounded-depth first search (with depth=8) to find small clusters (of a parameterizable size), then distribute these round-robin to different S-slices. It requires O(V) time.

Note the effect is proportional to the number of active vertices, hence high diameter graphs prefer load balanced multi-level as active vertices is usually low across iterations. For low diameter, larger clusters are helpful for locality.

## VII. Methodology

**PolyGraph Power/Area:** We prototyped PolyGraph by extending DSAGEN [46] with task scheduling hardware, and extended the stream-dataflow [31] ISA as described. We synthesized PolyGraph cores and NoC at 1GHz, with a 28nm UMC library. We used Cacti 7.0 [30] for modeling eDRAM.

**Baseline Architectures:** For reference, we use a 24-core SKL CPU and GAP benchmarks [4]. For CF and GCN (unavailable

[5]Grid-graphs are somewhat representative of common road networks.

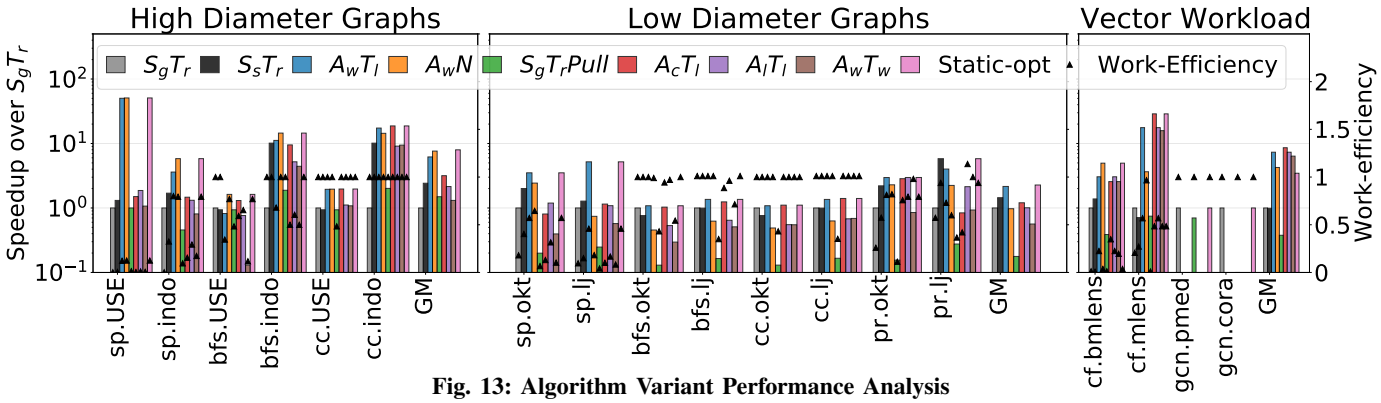



**Fig. 13: Algorithm Variant Performance Analysis**

in GAP), we used Graphmat [39] and Gunrock [43] respectively. We evaluated Gunrock [43] graph processing library on a Titan V GPU. Gunrock does not implement CC/CF, so we calculate GPU means without these workloads.

For performance modeling across variants, we developed a custom cycle-level modular simulator. Main memory is modeled using DRAMSim2 [36]. Accelerator configurations are in Table III, and have similar memory capacity, bandwidth, and max throughput. We assume preprocessing is done offline and reused across queries. Note that both temporal partitioning (chunk-based Gemini [59]) and spatial partitioning are O(V).

For fairness and consistency with our simulation framework, we make the following provisions for prior accelerators: For Graphicionado [16], we did not implement capacity optimizations like extended graph slicing and coarsened edge table. For Ozdal [32], we bypassed the sequential consistency module; it is not required on our workloads. For Chronos [1], we modeled priority-order speculative execution with their no-rollback optimization. Since SLOT requires a single read-write object per task, we implemented the pull variant; this allows vertex granularity tasks instead of fine-grained edge tasks in push. We used PolyGraph's NoC, as Chronos is FPGA-based. For GraphPulse [35], we model their task coalescing, but for consistency with our simulator, we used distributed scheduling.

For Polygraph (PG) we evaluate: 1. PG-singleAlg: fixed-variant configuration that provides the best geomean speedup. 2. PG-multiAlg: Here flexibility is incrementally added for per-workload (static) and per-phase (dynamic).

**Datasets:** Table IV summarizes input graphs. For SSSP on unweighted graphs, random weights are assigned from [1:256). Due to prohibitive simulation times of exploration, we evaluate large graphs (twitter, rdUSW) only for overall performance.

**Workloads:** Table I lists the evaluated workloads. For graph-based machine learning workloads, PR and CF, we optimize for convergence by choosing a different learning rate for each algorithm variant (higher for asynchronous variants). For GCN inference, we only implement the graph synchronous variant, as asynchronous benefits are minimal due to GCN's non-converging behavior.

## VIII. Evaluation

Our objective is to evaluate how much and which kinds of flexibility are useful, across graph and workload types. First, we analyze algorithm variants (§VIII-A) and compare against prior accelerators (§VIII-B). Then, we discuss sensitivity to algorithm and hardware parameters. (§VIII-C,§VIII-D).

### A. Algorithm Variants Performance Comparison

Figure 13 compares *strong algorithm variants* – those which perform well on at least one workload/graph type – in terms of throughput and work-efficiency. Overall, we find that asynchronous-sliced, $A_wT_l$, is the optimal variant (3.72× geomean speedup over typical $S_gT_r$), while static flexibility can further improve by 1.29× for high diameter graphs[6].

*High Diameter Graphs:* For ordered workloads like SSSP, high diameter graphs are highly sensitive to asynchrony and priority ordering, as it helps to identify more useful paths. Therefore, the speedups over $Sync_{graph}$ is high. Among vertex-scheduling schemes, $Async_{creation}$/$Async_{locality}$ provides only modest work-efficiency. In addition, the synchronization overheads of synchronous variants (eg. $S_g$, $S_s$) or slicing (eg. $A_wT$), significantly hurt high diameter graphs. Overall, ($A_w$No-slice) performs best/similar for all scenarios.

*Low Diameter Graphs:* With priority ordering, the ordered workloads see work-efficiency benefits (`sp.orkut`, `sp.lj`). Since low diameter graphs have significant reuse due to power-law degree distribution, slicing greatly improves performance. The synchronization overhead is minimal due to many active vertices in each iteration. For PageRank, which has a dense frontier, $Sync_{slice}$T-$slice_{robin}$ provides speedups through memory efficiency. Overall $Sync_{slice}$T-$slice_{robin}$ and, $Async_{work\text{-}eff}$T-$slice_{robin}$ are sufficient.

*Vector Workloads:* With asynchrony, collaborative filtering sees high gains, however priority scheduling is not required. For `bmlens`, the graph does not fit on-chip, and non-sliced is better as cross-slice updates are more critical here. Moreover, large vertex properties have high spatial locality that reduces cache miss overhead.

**Less Competitive Variants:** We generally find that with sufficient hardware for priority-order asynchronous scheduling, synchronous with sliced-work-efficiency is not competitive. It

[6]The overall potential is even higher for larger graphs, discussed in §VIII-B.

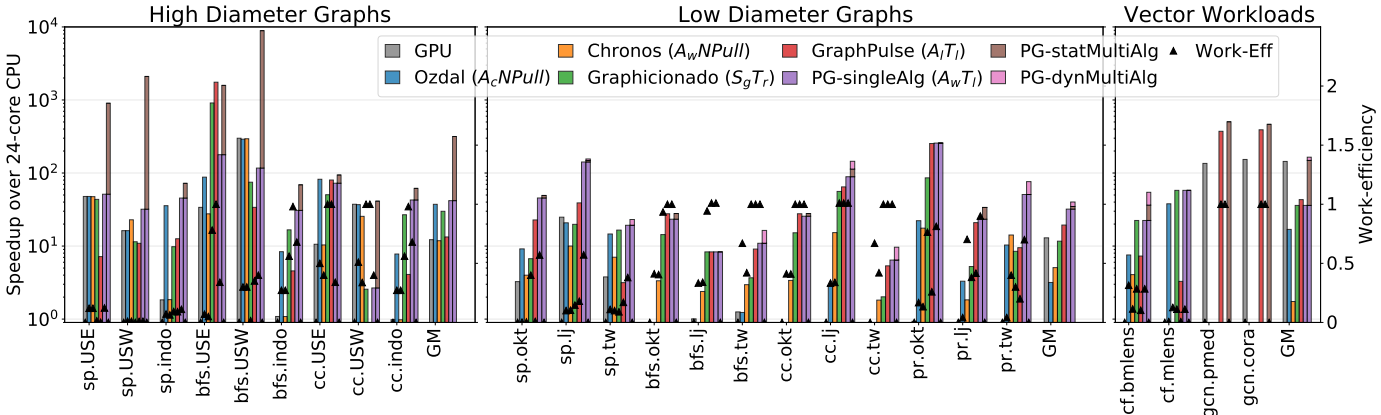



**Fig. 14: Overall Performance Comparison** (Gunrock does not implement CC, CF; GCN does not have pure asynchronous implementation.)

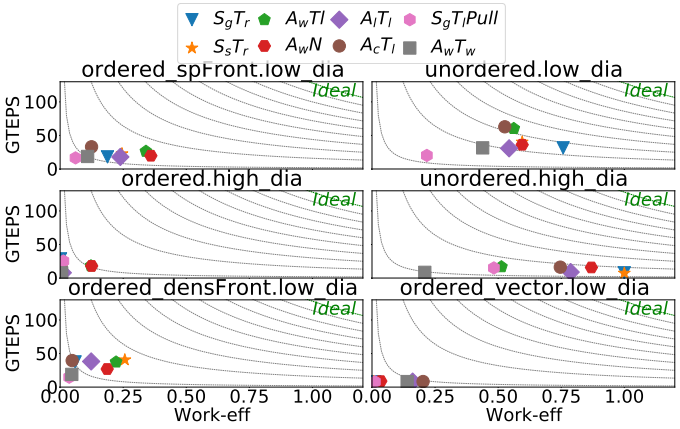



**Fig. 15: Utilization and Work-efficiency Tradeoff**

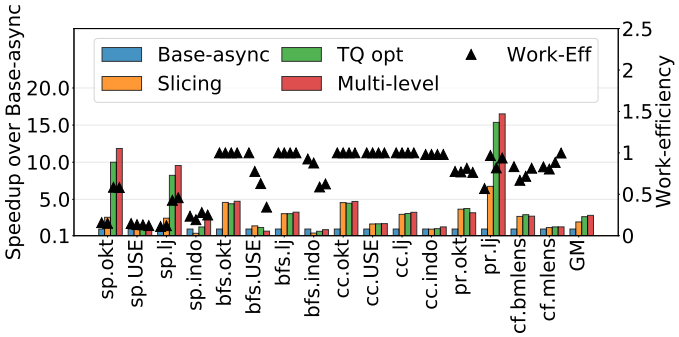



**Fig. 16: Cumulative Speedup of Novel Features**

also does not improve performance for asynchronous variants, as they require less iterations due to dynamic task creation.

Finally, the best pull variant, $S_gT_r$Pull with break optimization [3], consistently performs worse due to pipeline stalls waiting on random reads and work-efficiency loss from accessing all incoming edges irrespective of whether they are active. It is only competitive with its push counterpart for the `USE` graph, as it is 1. undirected: so no/less extra computation, 2. easy-to-partition: random reads are mostly local.

**Work-efficiency vs Throughput for Algorithm-Variants:** Figure 15 further explains the workload and graph type tradeoffs. Slicing improves memory efficiency for low diameter graphs, while $A_w$ helps in improving work-efficiency for order-sensitive workloads. Since high diameter graphs are regular, No-slice is superior as it achieves high hit rate while both avoiding barrier overheads and optimizing for work-efficiency. For dense frontier workloads, slice synchronous balances memory and work-efficiency. For the vector workload, CF, memory efficiency is implicitly high, thus asynchronous variants dominate due to faster convergence.

### *B. Comparison to Prior Accelerators*

Figure 14 shows the overall comparison. PG-statMultiAlg allows variant flexibility at the workload level, and PG-dynMultiAlg enables dynamic switching. Overall, PolyGraph outperforms CPU by 105.7× and GPU by 49.4×. Over the fastest prior accelerator, GraphPulse, it achieves 5.7× speedup. *High Diameter Graphs:* PolyGraph achieves work-efficiency-proportional gains over Gunrock's $\text{Sync}_{\text{graph}}$ GPU version. Even through Chronos and Ozdal use non-sliced implementations, they also use the inefficient pull variant. For the `USW` graph that does not fit in on-chip memory, GraphPulse and PG-singleAlg lose due to slicing overheads of switching time and work-efficiency loss due to delayed cross-slice vertices. A non-sliced variant can avoid these overheads.
*Low Diameter Graphs:* For order-sensitive SSSP and PR, PG-singleAlg gains work-efficiency due to vertex scheduling and slicing significantly improving hit rate. Since BFS and CC are less order-sensitive, PG-singleAlg behaves similar to Graphicionado. However, dynamic switching improves performance (eg. `bfs.lj,bfs.tw`).
*Vector Workloads:* For GCN, accelerators provide high speedup, as they support efficient broadcast of weight matrices, while GPU is bottlenecked by the unified cache bandwidth [43]. PolyGraph gets about 25% speedup by overlapping the communication-intensive aggregation and computation-intensive multiplication tasks. CF gains work-efficiency with synchrony and throughput with switching to the non-sliced variant in later iterations.

**Novel Features:** Figure 16 shows the cumulative speedup of each novel features over an asynchronous-push accelerator. In general, slicing significantly improves memory efficiency on low diameter graphs (`okt,lj`), while it causes slowdown in high diameter graphs due to work-efficiency loss (`bfs.indo`). Task coalescing particularly benefits low

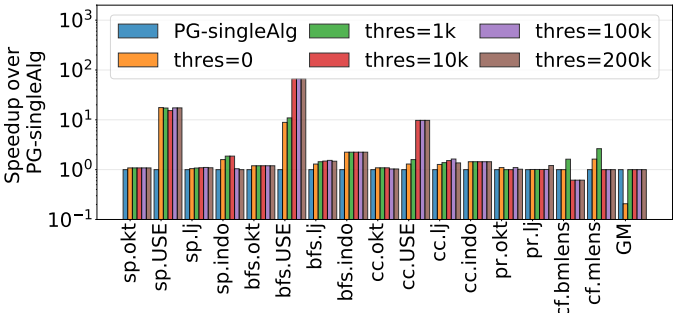



**Fig. 17: Dynamic Switching and Threshold Sensitivity**

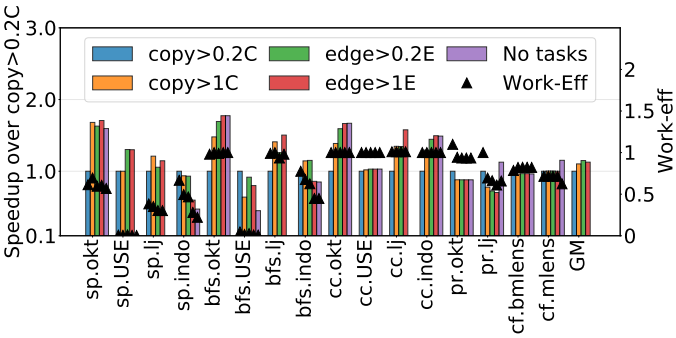



**Fig. 18: Slice Switch Heuristics** (C: cross-slice vert., E: edges/slice)

diameter graphs by eliminating superfluous updates to high degree vertices. The locality optimized multi-level partitioning with a fixed cluster size provides 20% benefit. For `BFS.USE`, multi-level is worse than naive partitioning because load is the primary bottleneck when locality is available. Thus, optimal cluster size depends on the input.

### C. Algorithm Sensitivity

**Dynamic Switching Heuristic:** Figure 17 shows the results of dynamic switching, with the heuristic described in §IV-C.

Here, we compare performance as we sweep the switching threshold of active vertices. The initial variant is the single-variant-optimal ($A_wT_l$). For low diameter graphs (`lj,okt,mlens`), performance improves to some point due to avoiding slice-switch overheads (up to 28% here); after this point performance reduces due to low memory efficiency of non-sliced. The effect is more dominant on non-computation intensive workloads like BFS. For high diameter graphs, the speedup plateaus, as non-sliced can achieve similar memory efficiency; thus, static flexibility is sufficient for them. Overall, dynamic switching helps primarily low diameter graphs.

**Slice Switching Heuristic:** Figure 18 compares slice switching heuristics on order-sensitive workloads[7]. "No tasks" represents switching when no outstanding tasks are left; it is either worse performance (`sp.indo,bfs.indo`) or does not converge (`sp.use,sp.lj`). Low diameter graphs (`okt,lj,mlens`) prefer a larger threshold because their clustered structure allows higher ratio of intra-slice vs inter-slice updates. High diameter graphs (`use,indo`) are highly sensitive to delayed updates and prefer to switch earlier. Note that for larger high diameter graphs, the work-efficiency loss still

[7]Note that since `okt,USE` fits in PolyGraph's on-chip memory, we reduced the on-chip memory size to half to make this experiment interesting.

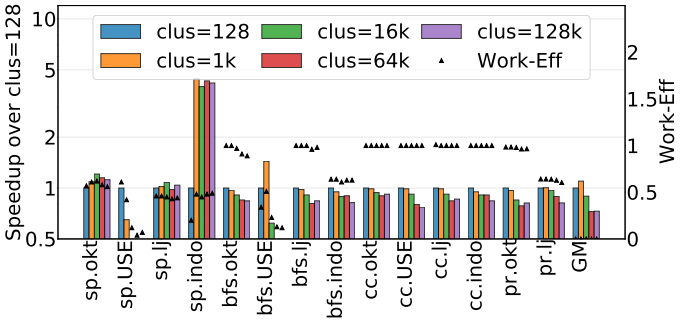



**Fig. 19: Sensitivity to Spatial Partitioning Cluster Size**

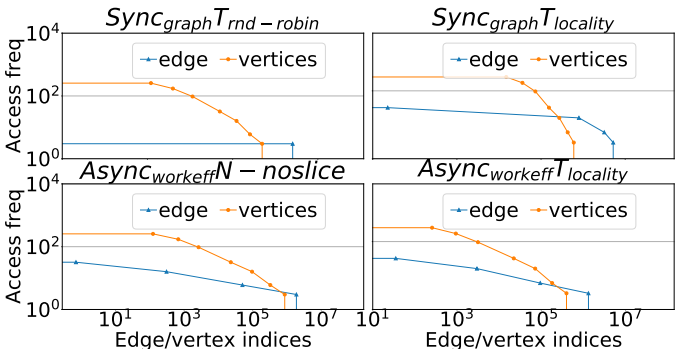


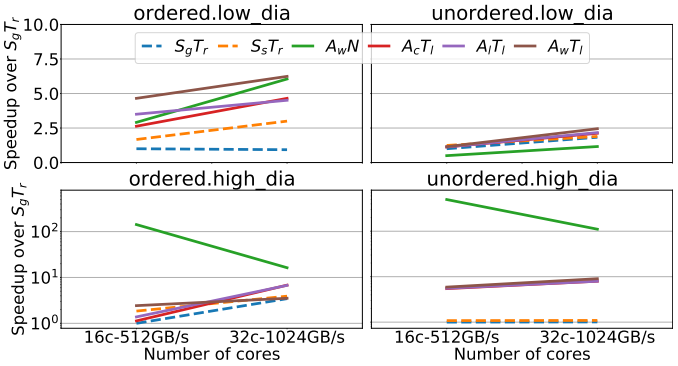

**Fig. 20: Access Patterns in Algorithm Variants** (for SP.lj)

dominates, and no-slicing wins if flexibility is available. We chose the edge-based heuristic, with a slicing threshold of 0.25*E for high diameter graphs and 1*E for low diameter graphs.

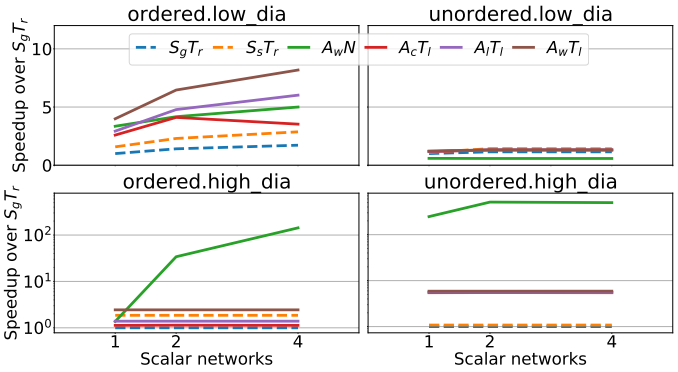
**Spatial Partitioning:** Figure 19 evaluates the multi-level spatial partitioning policy for different cluster sizes (on $A_wT_l$ variant)[8]. The results suggest small clusters optimize for dynamic load balance, while larger clusters improve locality. SSSP has higher computation intensity, and is thus more bottlenecked by locality than load balance; hence larger cluster sizes are better. BFS prefers smaller cluster sizes as memory level parallelism is more critical due to its low computation intensity. The default cluster size is 128; we use 16k for low diameter graphs (except BFS). Overall, flexible multi-level spatial partitioning provides 40% performance gain over conventional clustering.

**Per Data-structure Reuse:** Figure 20 shows the per-phase access frequency of edge and vertex data-structures for a subset of interesting variants. The access counts are averaged for a single phase for synchronous or 100k cycles for asynchronous. In general, vertices are more critical to cache on-chip because they have higher reuse and also require finer-grained random accesses. Although edge reuse is also significant because of iterating over a single slice multiple times (T-slice$_{\text{locality}}$ variants), we found that caching edges is not always beneficial. This is because it reduces vertex slice size too much, which hurts work-efficiency. Finally, the vertex_list data-structure has similar access behavior as vertex_prop, therefore we pin vertex_prop and vertex_list on-chip for sliced variants.

[8]We do not evaluate this for CF, GCN because, due to their large vertex properties, their maximum cluster size is too small.




**(a) Cores and Memory-bandwidth**




**(b) Number of 8-byte Networks**

**Fig. 21: Sensitivity to Hardware Resources**

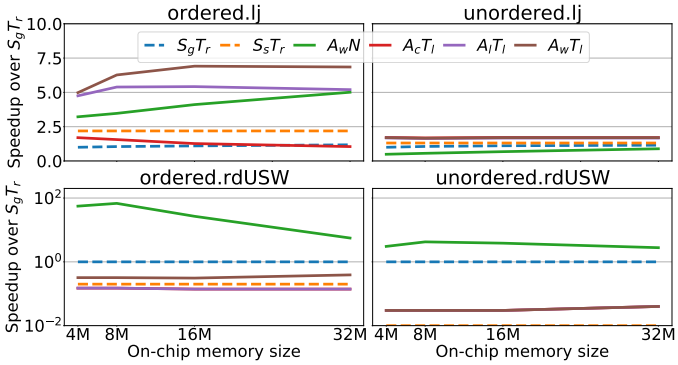



**Fig. 22: Sensitivity to Memory Size**

| | Area ($mm^2$) | Power (mW) |
|---|---|---|
| Control Cores | 0.053 | 11.5 |
| Priority Task Queue | 0.05 | 15.86 |
| Task Coalescer | 0.05 | 3.4 |
| on-chip mem+ctrl | 4.15 | 25.64 |
| CGRA (4x5) | 0.21 | 80 |
| 8x8 8-byte crossbar | 0.002 | 1.92 |
| **1 PG-Core** | 4.51 | 138.3 |
| 4x4 32 byte mesh (1) | 0.2 | 44.7 |
| 4x4 8 byte mesh (3) | 0.2 | 34.22 |
| **PG Total** | 72.56 | 2292.12 |

**TABLE V: Area and Power breakdown for PG-flex (28nm)**

### *D. Hardware Sensitivity*

In this section, we discuss the performance sensitivity of graph algorithm variants to hardware resources. Note ordered workloads are geomean of SSSP, PR and CF and unordered include BFS and CC. We discuss GCN separately.

**PolyGraph Scaling:** With 2x cores (and 2x memory bandwidth), the performance scales well (limited by available parallelism) as shown in Figure 21(a). Even though low diameter graphs are highly sensitive to memory bandwidth, scaling on unordered workloads is limited by parallelism while ordered workloads suffer due to loss in work-efficiency with larger network latency. Since high diameter graphs are easy-to-partition, they ensure high hit rate and thus reduced dependence on memory bandwidth. $A_wN$ `high_dia` case shows performance loss with more cores: this is because larger working set means higher sensitivity to factors affecting work-efficiency, like larger network latency for 32 cores.

**Network bandwidth:** Figure 21(b) sweeps over scalar network bandwidth. The sliced variants of low diameter graphs have good on-chip memory locality, and are bottlenecked by network bandwidth. For high diameter graphs, both sliced/non-sliced variants achieve high hit rate and memory locality. Therefore, sliced variants are bottlenecked by load imbalance due to frequent synchronization, while for non-sliced, work-efficiency is proportional to the network bandwidth.

**On-chip memory size:** Figure 22 shows the performance sensitivity to on-chip memory size: this is a proxy for scaling up graph-size (more slices required and more cache pressure), while using a consistent input. The data is presented for specific input graphs as the saturation point depends on the ratio of graph and on-chip memory size. For ordered workloads, a larger memory size improves work-efficiency with lesser cross-slice edges while reducing the required number of barriers. The latter is a small factor, as can be seen for `unordered` cases. `Ordered.rdUSW` $A_wN$ is an exception where performance degrades with larger on-chip memory; this happens because a larger working set may cause cores to become too unsynchronized, hurting working efficiency.

**GCN Sensitivity:** When doubling core count, GCN's performance scales by 1.9×, with little loss due to load imbalance in aggregation phase. For scaling network bandwidth, GCN improves linearly up to 64-byte bandwidth, after which computation becomes the bottleneck. For scaling down on-chip memory, only if we scale down to 4MB does the memory bandwidth become the bottleneck; this happens with 7 slices, since our GCN's graphs require maximum 28 MB.

**Area Tradeoffs:** Table V shows PolyGraph's area breakdown. It occupies 72.56$mm^2$, with eDRAM consuming 91.1% of the total area. Compared to Graphicionado, PolyGraph is similar area with 84% power due to using a mesh instead of a crossbar.

Figure 23 compares accelerator speedup and area. Overall, PolyGraph has similar area as Graphicionado while achieving 7.2× speedup due to its optimizations and flexibility. We also examine area tradeoffs for PolyGraph by removing the components that consume significant area (eliminating certain variant options). Without caches, memory flexibility is not available, hurting high diameter graphs. Without a priority queue, the gains on order-sensitive workloads is reduced. With no dynamic tasks, $S_sT_r$ is the best variant, as it provides

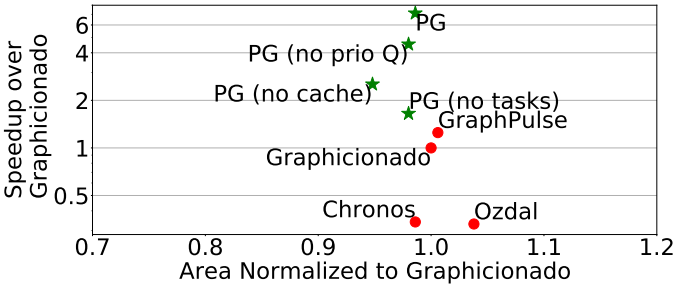



**Fig. 23: Accelerator Performance vs Area**

some work-efficiency by conveying updates sooner, with high memory efficiency of locality scheduling.

## IX. Additional Related Work

Figure VI categorizes prior work by variant. All variant combinations are supported by PolyGraph.

**Graph Frameworks with Flexibility:** While some software graph frameworks focus on a single algorithm variant (eg. GraphMat [39]: $S_gN$), several others allow certain amount of flexibility in their programming model. For example, Galois [33] provides optional support for scheduling vertex buckets by a data-dependent priority (Minnow [52] provides hardware support for CPUs). Salvador et. al. [37] studies the interaction of update direction and memory coherence/consistency models for GPUs, and demonstrates the usefulness of flexibility. We further show the usefulness of flexibility across dimensions of update visibility and vertex/slice scheduling.

Powergraph [14] supports both synchronous and asynchronous variants. Powerswitch [48] adds heuristics to switch between sync/async dynamically, which we find is not effective for accelerators. X-Stream is "edge-centric", where edges are streamed without sparse access through vertex indices. Even though edge-centric can be supported by PolyGraph, we did not consider it as it is incompatible with key optimizations like priority ordering and vertex-based dynamic tasks.

**Graph Taxonomies:** McCune et al. classified distributed graph frameworks [27]. Lenharth's taxonomy [24] identifies factors that impact graph execution: (topology, synchronicity, reordering, graph operators). These works do not consider hardware specialization or slicing.

**Hardware Accelerators:** Graphicionado [16] accelerates push-based synchronous variant. GraphDyns [50] adds dynamic work-distribution. Ozdal et. al. supports sequential consistency in its asynchronous graph processing ASIC template [32]. Our evaluated workloads do not require sequential consistency guarantees for correctness; only CF may converge faster with consistency [25].

Digraph [56] is a multi-GPU system for asynchronous graph processing. Chronos [1] is the only prior asynchronous accelerator that supports fine-grained priority scheduling. Several designs exploit multiple HMC nodes (eg. Tesseract [2]). GraphP [54] extends Tesseract with a two-phase programming model enabling efficient partitioning. GraphQ [61] has a hybrid execution model; asynchronous within each HMC.

| | Non-sliced(N) | Temporally-sliced (T) |
|---|---|---|
| $Sync_{graph}(S_g)$ | Tesseract [2] | Graphicionado [16] ($T_{robin}$) |
| | GraphMat* [39] | GraphDyns [50] ($T_{robin}$) |
| $Sync_{slice}(S_s)$ | | GraphQ* [61] ($T_{robin}$) |
| | | GraphABCD* [51] ($T_{work-eff}$) |
| $Async_{local}(A_l)$ | Ozdal [32] | GraphPulse [35] ($T_{locality}$) |
| | Giraph* [17] | |
| $Async_{creat.}(A_c)$ | Graphlab* [25] | |
| $Async_{Work}(A_w)$ | Chronos [1] | Digraph* [56] ($T_{work-eff}$) |
| | Galois* [33] | |
| | Minnow [52] | |

**TABLE VI: Prior Works in Taxonomy** (*software frameworks)

DepGraph [57] combines updates across frequently accessed paths, reducing re-execution overhead. This is an alternative way of improving work-efficiency.

**Graph Spatial Locality Techniques:** Graph preprocessing is employed to improve spatial locality [13,20,40], which is especially useful if graphs are executed in locality/vertex order. HATS [28] is a CPU offload accelerator which dynamically discovers graph locality. Polymer [53] explores spatial placement and replication of vertices in a distributed system.

## X. Conclusion

This work studies the value of flexibility in graph processing accelerators by systematically analyzing codesign tradeoffs along graph algorithm variants. Broadly, we find that flexibility is essential for even a modest range of graph workloads. Specifically, we find that both synchronous/asynchronous and priority-based/locality-based vertex scheduling are critical for balancing work-efficiency and locality. Also, dynamically switching between sliced/non-sliced variants across workload phases enabled further specialization for low diameter graphs.

To support all algorithm variants with accelerator-level performance, including novel combinations, we developed extensions to a dataflow model which embeds first-class primitives for dynamic parallelism and designed a specialized task management and memory system. The principles developed here may be helpful in broader workloads requiring fine-grain task scheduling and management of large working sets.

## XI. Acknowledgments

This work was supported by NSF awards CCF-1751400 and CCF-1937599, as well as gift funding from VMware.

## References

[1] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," ser. ASPLOS '20, 2020.
[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.
[3] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," ser. SC'12, 2012.
[4] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.
[5] L. Belayneh and V. Bertacco, "GraphVine: exploiting multicast for scalable graph analytics," in *DATE*, 2020.
[6] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *INFOCOM*, 2000.
[7] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (RSVP)," ser. MICRO 36, 2003.

[8] J. Cong, M. A. Ghodrat, M. Gill, C. Liu, and G. Reinman, "Bin: a buffer-in-NUCA scheme for accelerator-rich CMPs," in *ISLPED*, 2012.

[9] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of CGRA," in *FCCM*.

[10] N. Corp, "GeForce GTX 1080 Whitepaper."

[11] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," ser. MICRO '52, 2019.

[12] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "GraphH: a processing-in-memory architecture for large-scale graph processing," *IEEE TCAD*, April 2019.

[13] T. A. Davis, W. W. Hager, S. P. Kolodziej, and S. N. Yeralan, "Algorithm xxx: Mongoose, a graph coarsening and partitioning library," *ACM Trans. Math. Software*, 2019.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.

[15] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, Sep. 2012.

[16] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*, Oct 2016.

[17] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proc. VLDB Endow.*, May 2015.

[18] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-centric execution of speculative parallel programs," in *MICRO*, Oct 2016.

[19] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *MICRO*, Dec 2015.

[20] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, Dec. 1998.

[21] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: Media processing with streams," *IEEE micro*, 2001.

[22] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[23] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *ASPLOS*, ser. ASPLOS '18, 2018.

[24] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Communications of the ACM*, 2016.

[25] 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," *Proc. VLDB Endow.*, Apr. 2012.

[26] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *SIGMOD*, ser. SIGMOD '10, 2010.

[27] R. R. McCune, T. Weninger, and G. R. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for distributed graph processing," *CoRR*, 2015.

[28] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *MICRO*, 2018.

[29] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: architectural support for synchronization- and bandwidth-efficient commutative scatter updates," ser. MICRO '52, 2019.

[30] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009.

[31] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," ser. ISCA '17, 2017.

[32] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *ISCA*, June 2016.

[33] 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," ser. PLDI '11, 2011.

[34] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," ser. ISCA '17, 2017.

[35] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "GraphPulse: an event-driven hardware accelerator for asynchronous graph processing," in *MICRO*, 2020.

[36] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: a cycle accurate memory system simulator," *IEEE CAL*, 2011.

[37] G. Salvador, W. H. Darvin, M. Huzaifa, J. Alsop, M. D. Sinclair, and S. V. Adve, "Specializing coherence, consistency, and push/pull for gpu graph analytics," in *ISPASS*, 2020.

[38] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," ser. PPoPP '13, 2013.

[39] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *VLDB*, 2015.

[40] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," ser. WSDM '13, 2013.

[41] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, 1990.

[42] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *CIDR*, 2013.

[43] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *PPoPP*, 2016.

[44] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," ser. ISCA '19, 2019.

[45] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream floating: Enabling proactive and decentralized cache optimizations," in *HPCA*, 2021.

[46] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "DSAGEN: synthesizing programmable spatial accelerators," in *ISCA*, 2020.

[47] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, "Scalable data-driven pagerank: Algorithms, system issues, and lessons learned," in *European Conference on Parallel Processing*, 2015.

[48] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: time to fuse for distributed graph-parallel computation," ser. PPoPP 2015.

[49] C. Xu, C. Wang, L. Gong, L. Jin, X. Li, and X. Zhou, "Domino: Graph processing services on energy-efficient hardware accelerator," in *ICWS*, 2018.

[50] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," ser. MICRO '52, 2019.

[51] Y. Yang, Z. Li, Y. Deng, Z. Liu, S. Yin, S. Wei, and L. Liu, "GraphABCD: scaling out graph analytics with asynchronous block coordinate descent," in *ISCA*, 2020.

[52] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," ser. ASPLOS '18, 2018.

[53] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *PPoPP*, 2015.

[54] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: reducing communication for PIM-based graph processing with efficient data partition," in *HPCA*, 2018.

[55] Y. Zhang, A. Rucker, M. Vilim, R. Prabhakar, W. Hwang, and K. Olukotun, "Scalable interconnects for reconfigurable spatial architectures," in *46th ISCA*, 2019.

[56] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu, "Digraph: An efficient path-based iterative directed graph processing system on multiple GPUs," ser. ASPLOS '19, 2019.

[57] Y. Zhang, X. Liao, H. Jin, L. He, B. He, H. Liu, and L. Gu, "Depgraph: A dependency-driven accelerator for efficient iterative graph processing," in *HPCA*, 2021.

[58] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *International conference on algorithmic applications in management*, 2008.

[59] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *OSDI*, 2016.

[60] Y. Zhuo, J. Chen, Q. Luo, Y. Wang, H. Yang, D. Qian, and X. Qian, "SympleGraph: distributed graph processing with precise loop-carried dependency guarantee," in *PLDI*, 2020.

[61] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," ser. MICRO '52, 2019.