This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MM.2022.3160862, IEEE Micro

# Systematically Understanding Graph Accelerator Dimensions and the Value of Hardware Flexibility

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. Most prior such accelerators adopt a single fixed algorithm. 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. Our approach is to identify a taxonomy of key algorithm variants, and develop a modular architecture, PolyGraph, which is flexible across them. The key to flexibility is our novel Taskflow execution model, which unifies task and dataflow parallelism. Overall we find that flexibility is essential; PolyGraph outperforms similarly provisioned GPUs by mean 49.6× (up to 275×), and the best prior accelerator by mean 5.7×.*

## 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 hardware specialization opportunities: commutative updates, resilience to work reordering, and repetitive structure in memory access and computation. This implies large advantages for hardware accelerators.

Often for simplicity, these designs make strong assumptions about certain aspects of graph processing and implement only a single fixed algorithm, which can limit their ability to generalize. This includes assuming a certain input graph type (eg. high vs. low diameter) or workload property (eg. order resilience, frontier density, computation intensity). Consequently, it is difficult to know when to apply which kind of accelerator, and what the value of flexibility in graph processing might be.

To conceptualize the fundamental differences between accelerators, we identify a taxonomy of four *graph algorithm variants* i.e. algorithmic dimensions that significantly influence hardware specialization: 1. Update Visibility: granularity when vertex updates become visible, 2. Vertex Scheduling: fine-grain scheduling policy for vertices, 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). As we will demonstrate, these variants

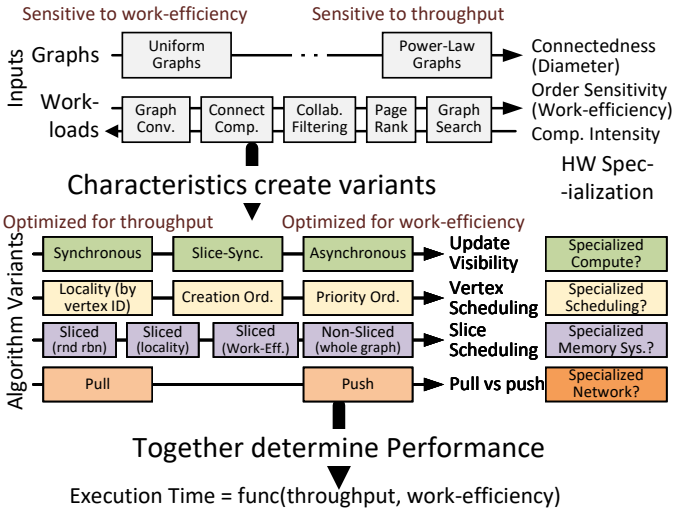


**Fig. 1: Relationship between Inputs, Variants and Performance.**

dramatically affect the tradeoff between throughput and work-efficiency (i.e. how close to the optimal amount of computation is performed), which are the two major factors that determine performance.

Figure 1 depicts how different graph and workload properties influence algorithm variant choices, and how these both influence the throughput versus work-efficiency tradeoff. Each variant has profoundly different implications on hardware codesign, as shown to the right of each variant in the figure.

We answer two key questions: First, to better understand the space of graph processing techniques, what is the relationship between different dimensions in the taxonomy and their overall effect on performance? Second, what is the value of flexibly supporting multiple algorithm variants? Our approach is to create a modular execution model and accelerator which can optionally support the hardware features necessary for each variant, while limiting area, power, and performance overheads. This flexibility requires supporting broad data-structures, different task granularities (synchronous vs asynchronous updates), fine-grain task scheduling, and working set control.

Our design augments efficient decoupled-spatial accelerators [4,6], which support general data-structures and suits both memory-intensive (eg. Breadth-first Search (BFS)) and compute-intensive workloads (eg. Graph Convolutional Networks (GCN)). The fundamental limitation of prior accelerators is the lack of support for fine-grain data-dependent

parallelism (i.e. task parallelism). Therefore, 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.

**Evaluation and Contribution:** We evaluated our accelerator, PolyGraph, with cycle-level simulation, supporting an architecture design space with features encompassing many prior works [1,5,7,10]. We studied traditional and ML-based graph workloads on real-world graphs. The best fixed-algorithm PolyGraph design is 16.79× faster than a Titan V GPU (up to 275× for high diameter graphs). More importantly, the variant flexibility provides 2.95× speedup.

Our work has two primary contributions. The first is the graph algorithm taxonomy and observations that relate graph accelerator dimensions and characterize the value of flexibility. The second is the unification of task and dataflow execution models in Taskflow, along with the novel microarchitectural support within PolyGraph.

## II. Graph Accelerator Design Space

We first overview graph and workload properties that creates the need for flexibility. Then, we describe the vertex-centric computational paradigm and how its specializable properties give rise to graph algorithm variants.

### A. Background: Graph and Workload Properties

Figure 2(a) visualizes graph and workload types. We define them below:

**Graph Property – Diameter:** The 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, while *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*, and thus are resilient to work ordering of individual tasks. In some of these workloads, different task orderings are more or less efficient; ie. their work-efficiency is *order-sensitive*. For example, shortest path algorithms (SSSP) on graphs with a wide distribution of edge distances are order-sensitive.

**Workload Property – Frontier Density:** Dense frontier workloads like PageRank (PR), Collaborative Filtering (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.

### B. Graph Algorithm Variants Taxonomy

We rely on the vertex-centric execution model as the starting point for defining algorithm variants, as shown in Figure 2(b1). Here, the programmer may (or may not) split the input graph into multiple temporal slices (T-slices), such that each slice fits into on-chip memory (Figure 2(b2) shows execution over T-Slices). Each T-slice is further split into spatial slices (S-slices), across cores. Within an S-slice, the computation is performed at the vertex granularity: For each active vertex, its outgoing edges are accessed, then these destination vertices are updated using user-defined functions (see Figure 2(b1-i, b1-ii, and b1-iii)). The vertices are conditionally activated until convergence (Figure 2(b1-iv)). We identify four critical *graph algorithm variants* dimensions (shown in Figure 2(c)) with tradeoffs in Figure 2(d).

**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*) or immediately (*asynchronous*). Barriers are used to synchronize update propagation in synchronous variants. Figure 2(c1) visualizes how dependence distance (red arrows) shrinks when moving from synchronous to asynchronous.
*Tradeoffs:* The static nature of synchronous algorithms makes it easier to optimize for efficient memory access, while the lower dependence distance of asynchronous variants leads to faster convergence while avoiding barriers.

**Vertex Scheduling:** defines the processing order for active vertices for asynchronous variants. Figure 2(c2) depicts the variants for shortest path: *Locality order:* In order of vertex-id. *Creation order:* In the order tasks are created. *Work-efficiency order:* In the order of least redundant work (by distance in the figure).
*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.

**Temporal Slice Scheduling:** If using temporal slicing, slices can be scheduled in different orders, forming new variants. Figure 2(c3) depicts each: *Round-robin:* iterate through all slices. *Locality:* similar, but repeatedly process each slice. *Work-efficiency:* prioritize slices whose properties change most [12]. 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 leads to more effective on-chip memory use but can harm the optimal ordering by restricting the scheduling scope. 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, rather than the remote memory requests in pull. Also, pull often requires more work while reading all incoming edges of each active vertex.

**Not included Variants:** We did not explore all dimensions

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MM.2022.3160862, IEEE Micro

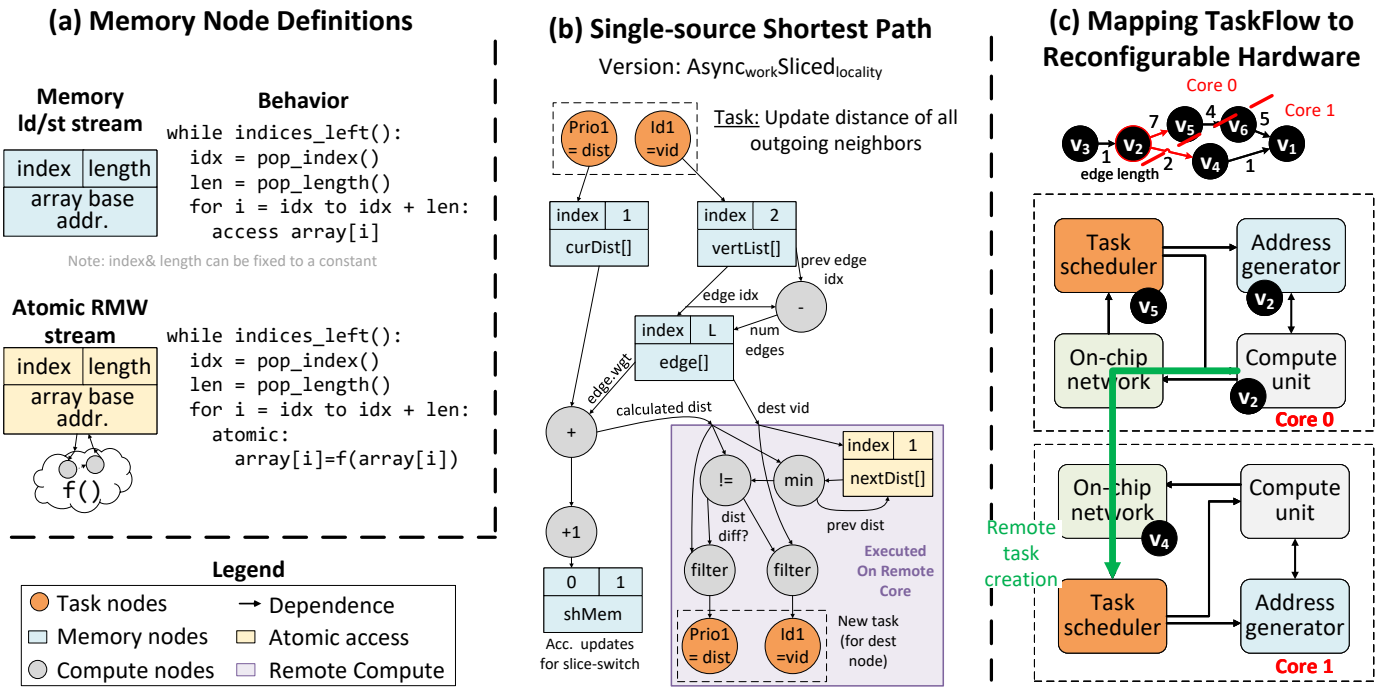


**Fig. 2: Key Variants of Graph Processing Algorithms**

of graph processing as the space is very large, and some optimizations are more narrowly useful: 1. Edge-centric paradigm, where edges are streamed without sparse access through vertex indices. It is incompatible with key optimizations like priority ordering and vertex-based dynamic tasks. 2. Complex scheduling, where certain traversal information is memoized to reduce the required work (e.g. [12]). 3. Dynamic graph partitioning, where slice-assignment changes dynamically, which could be useful when the graph is modified dynamically. 4. Processing-in-memory Support: use of bit vector or resistive memory crossbars for processing-in-memory.

## III. Taskflow: Unifying Task Parallelism and Dataflow Execution

A flexible graph processing accelerator should provide accelerator-like throughput while enabling task-parallelism. Specifically, the accelerator must: 1. Allow fully pipelined per-vertex computation. 2. Support high-throughput task creation and scheduling, and 3. Enable streaming, atomics and memory reuse.

To this end, we create a unified task-parallel and dataflow model called *Taskflow*. In taskflow, a task is invoked by its type `t` and input arguments. 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 [4,6] for high efficiency.
- **Memory nodes:** represent decoupled patterns of memory access, called streams [4,6]. Stream parameters can either be constant or dynamic (consumed from another node using a FIFO interface). Figure 3(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 ingress task node, 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 taskflow, a memory stream can be marked as "read-modify-write" (RMW) and will be atomic. See example in Figure 3a.

**Priority and Spatial Scheduling:** One task argument may be designated at compile time as the task's priority, which serves as a schedule-order hint. Another task argument is a unique ID, which indicates the task's temporal-slice and spatial-slice. Tasks are deferred until their temporal-slice is active and are scheduled at their spatial-slice's core.

**Taskflow Example:** Figure 3(b) shows an example taskflow graph for SSSP, implemented as Async$_{work}$.

*Taskflow Graph:* This workload has two tasks: Task type 1 iterates over outgoing edges of a vertex to compute distances, and creates a type 2 task for each destination vertex to carry out distance updates. The ID is the vertex-ID, and tasks execute on the corresponding core. 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 create a new type 1 task. The number of task 1 invocations is accumulated to determine when to switch slices (§IV).

*Mapping to hardware:* To demonstrate taskflow on hardware, consider a simplified multi-core view in Figure 3(c). Here the input graph is partitioned across two cores. Each task can be in the: 1. network waiting for remote creation at its assigned core (v4 at Core 1), 2. task scheduler before execution (v5), or 3. compute and memory unit during execution (v2).

**Spatial Partitioning:** For multicore taskflow to be effective,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MM.2022.3160862, IEEE Micro

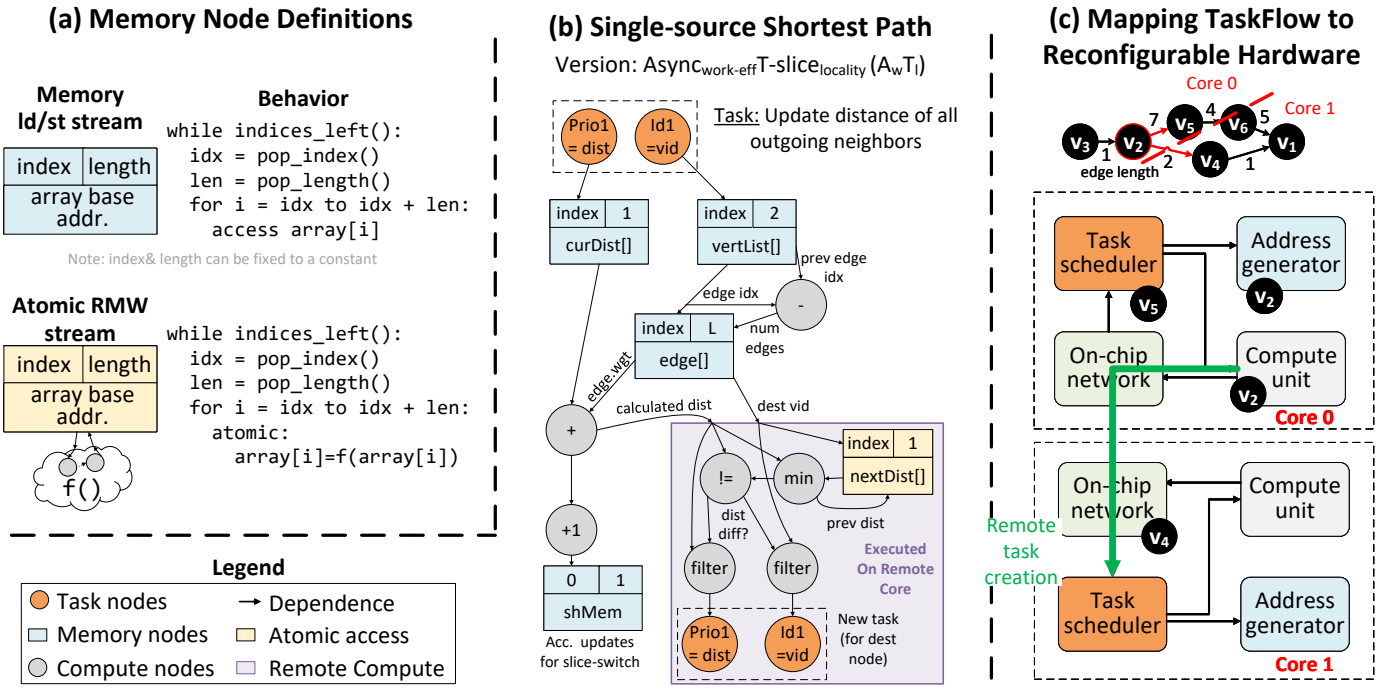


**Fig. 3: Taskflow Examples**

we spatially partition the graph. 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 propose a "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.

## IV. Slice Data Orchestration

While taskflow handles the cycle-by-cycle execution, the *slice scheduler* manages tasks and data for slice-based execution. As it is invoked infrequently, our implementation uses a simple control core with limited extensions. We overview its operation next, more details are in the original paper [3].

**Data Pinning:** Depending on the algorithm variant, we may know which data has the most reuse. For e.g. for graph-synchronous non-sliced variant, edges have a large reuse distance but vertices with high-degree are reused many times. In sliced-locality variants, edges have a smaller reuse distance. Thus, the slice-scheduler has an interface to *pin* a range of data to the on-chip memory at a particular offset, essentially reserving a portion of the cache.

**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). Information at the slice's boundary becomes "stale" over time, as it may depend on an inactive slice's execution, thus hurting work-efficiency. Therefore, we approximate "staleness" by counting the number of vertex updates, and switch slices when these exceed a threshold. This is carried out by giving all cores a highest priority stop task, to interrupt task execution.

**Temporal Slicing and Slice Transition:** For sliced execution, graphs are preprocessed to keep updates within a T-slice. Specifically, the source vertex of any cross-slice edge is replaced with a mirror vertex in the destination slice. The mirror properties are only updated during slice transition.

Here the main memory contains vertex properties, pending tasks for each slice, and a copy of each mirror vertex. During slice transition, old slice's information is sent to memory and the new slice's information is loaded to pinned memory. If a mirror's property is detected to have changed, a new task is created in the destination slice.

**Algorithm Variant Selection:** Variant selection is performed after each slice, or after every 100k cycles for asynchronous variants. To transition, the control core will initialize data-structures, configure taskflow graphs, and perform pinning operations. On-chip memories may need to be flushed, and taskflow may require reconfiguration. The pending tasks are managed just as they would be during slice transition. The following heuristics decide the next algorithm variant:

1) **Update visibility:** Asynchronous versions are preferred for order-sensitive workloads. Dense frontier algorithms have high inherent spatial locality, so $\text{Slice}_{\text{sync}}$ is preferred, as it is memory efficient while maintaining moderate work-efficiency.
2) **Temporal Slicing:** The effective throughput depends on whether the work during phase is sufficient to hide barrier

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MM.2022.3160862, IEEE Micro

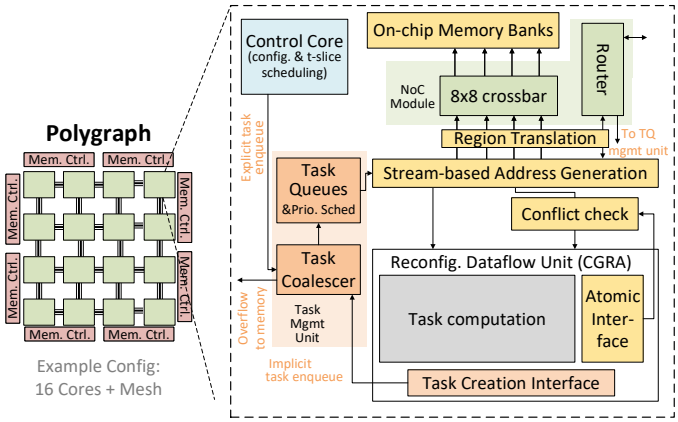


**Fig. 4: PolyGraph Modular Hardware Implementation**

overhead. This work can be approximated from active vertices. Therefore, our algorithm switches to non-sliced when number of active vertices are below a threshold (e.g. usually active vertices are low for high diameter graphs).

3) **Spatial Partitioning:** High diameter graphs prefer load balanced multi-level partitioning (i.e. smaller clusters) as connectivity is regular. For low diameter, larger clusters are helpful for locality.

## V. PolyGraph Hardware Implementation

PolyGraph is a multicore decoupled-spatial accelerator connected by mesh networks[1], overview in Figure 4. We first give background on decoupled-spatial accelerators and integration with task management. Then, we explain the design of task management unit and shared memory for slice scheduling.

**Integration of Tasks with Decoupled-spatial Accelerator:** In a decoupled-spatial accelerator, memory nodes are maintained on stream address generators, and accesses are decoupled to hide memory latency. A Coarse-grained Reconfigurable Array (CGRA) [6] 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 the task management: A priority-ordered task queue holds waiting tasks. Task nodes define how incoming task arguments into the queue are consumed by the stream controller and CGRA.

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 are created by the control core.

**Task Management Unit:** This unit includes three main components: task management buffer, task scheduler and overflow buffer. We describe these next.

[1]Multiple networks enable efficient scalar remote accesses.

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 scheduler maintains pointers to the task argument buffer entries. 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 [2], with a throughput of one enqueue-dequeue every two cycles. For low degree graphs, this throughput is insufficient. Therefore, we use multiple priority-heaps per core, and alternate between them.

If the task queue is full, new tasks are pushed into a *overflow 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* (in on-chip memory).

**Slice Scheduling:** is implemented by the slice scheduler on core 0's control core, with support of a shared memory with data pinning and atomic update capability. PolyGraph cores communicate with the slice scheduler through shared memory atomics to coordinate phase completion.

Our *on-chip memory* is a shared address-partitioned cache, with multiple banks per PolyGraph core. The region translation unit maintains the mapping of virtual address ranges to pinned addresses. When the slice-scheduler in core 0 pins a memory region, the region translation unit in all cores are sent the base/bound and offset of the region. This causes some of the sets of the cache to be set aside for pinned data. Pinning a new data-structure flushes all cache regions in the pinned address range; this is only required during variants transition, i.e. at most twice in a program.

When *atomic update* requests are received from the local core or network, they are pushed to the pending atomic request 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. Methodology

**PolyGraph Power/Area:** We prototyped PolyGraph by extending DSAGEN [11] with task scheduling hardware, and extended the stream-dataflow [6] ISA. We synthesized PolyGraph at 1GHz, with a 28nm UMC library. We used Cacti 7.0 for modeling eDRAM.

**Baseline Architectures:** For reference, we use a 24-core SKL CPU running GAP benchmarks and a Titan V GPU using Gunrock graph processing library.

We compare against four prior accelerators: Graphicionado [5], Ozdal [7], Chronos [1], and GraphPulse [10]. We provision all accelerators with 32MB on-chip memory, 1024 FP units and 512 GB/s memory bandwidth, with the GPU having more FP units and memory bandwidth. All prior accelerators use crossbar network, except Chronos and PolyGraph which rely on a mesh.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MM.2022.3160862, IEEE Micro

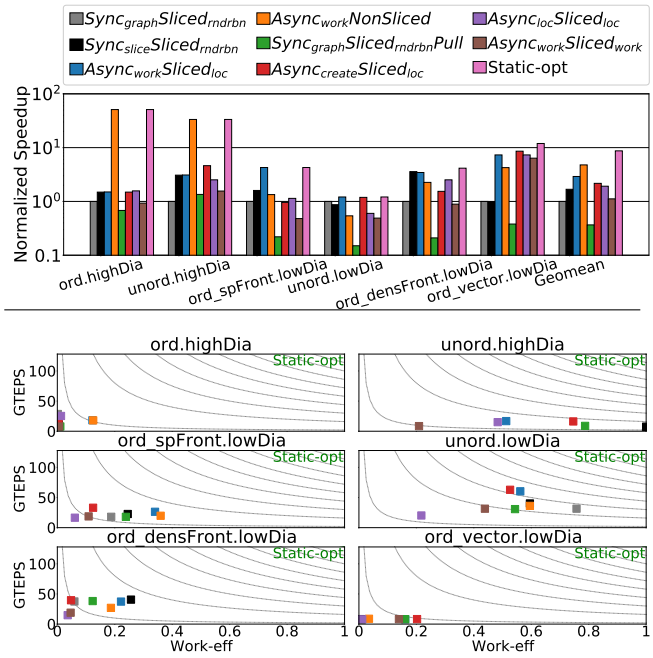


**Fig. 5: Comparison of Algorithm Variants.**

For accelerator performance modeling, we developed a custom cycle-level simulator where main memory is modeled using DRAMSim2. We assume preprocessing, only O(V), is done offline and reused across queries.

**Datasets:** We compare against practical graphs, categorized into low and high diameter.

**Workloads:** We study workloads in three categories: 1. Ordered with sparse frontier: SSSP, 2. Unordered with sparse frontier: BFS, and connected components (CC) 3. Ordered with dense frontier: PR, and 4. Ordered with vector computation: CF, and GCN.

## VII. Evaluation

Our evaluation broadly addresses the question of how much and which kinds of flexibility are useful, across graph and workload types.

**Algorithm Variants Comparison:** Figure 5 compares *strong algorithm variants* – those which perform well on at least one workload/graph type. Overall, we find that asynchronous-sliced, $\text{Async}_{\text{work}}\text{Sliced}_{\text{loc}}$, is the optimal variant (2.91× geomean speedup over typical $\text{Sync}_{\text{graph}}\text{Sliced}_{\text{rndrbn}}$), while static flexibility can further improve speedup by 3×. We explain the trends below, grouped by their choice of the best algorithm variant:

1) **High Diameter Graphs:** Here the synchronization overheads of synchronous/sliced variants (eg. $\text{Sync}_{\text{graph}}$, $\text{Sync}_{\text{slice}}$, $\text{Async}_{\text{work}}$Sliced) are the critical bottleneck. Therefore, $\text{Async}_{\text{work}}$Non-Sliced performs best/similar for all workloads.
2) **Low Diameter Graphs:** The power-law degree distribution makes random accesses more critical than synchronization. Sliced variants improve reuse, and thus perform better. For order-sensitive workloads (e.g. SSSP), faster updates in asynchronous variants lead to faster convergence. Among vertex-scheduling schemes, $\text{Async}_{\text{work}}$ performs best while $\text{Async}_{\text{creation}}$/$\text{Async}_{\text{locality}}$ provides only modest work-efficiency. Overall $\text{Async}_{\text{work}}\text{Sliced}_{\text{rndrbn}}$ is sufficient.
3) **Dense Frontier workloads:** For PageRank, which has a dense frontier, $\text{Sync}_{\text{slice}}\text{Sliced}_{\text{rndrbn}}$ provides speedups through memory efficiency while retaining some work-efficiency benefits of asynchronous updates within a graph slice.
4) **Vector Workloads:** With asynchrony, vector workload, CF sees high gains, however priority scheduling is not required. Non-sliced is similar to sliced as large vertex properties have high spatial locality that reduces cache miss overhead.

*Less Competitive Variants:* We generally find that with sufficient hardware for asynchronous priority scheduling, sliced-work-efficiency does not help as asynchronous variants require less iterations due to dynamic task creation.

Finally, the best pull variant, $\text{Sync}_{\text{graph}}\text{Sliced}_{\text{rndrbn}}$Pull 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.

**Work-efficiency vs Throughput for Algorithm-Variants:** Figure 5 further explains the workload and graph type tradeoffs. Slicing improves memory efficiency for low diameter graphs, while $\text{Async}_{\text{work}}$ improves work-efficiency for order-sensitive workloads. Since high diameter graphs are regular, Non-sliced is superior as it achieves high hit rate while avoiding barrier overheads. 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.

**Area Tradeoffs:** PolyGraph occupies 72.56mm$^2$, with eDRAM consuming 91.1% of the total area. The task management structures occupy 2% of the total area. Compared to Graphicionado, PolyGraph is similar area with 84% power due to using a mesh instead of a crossbar.

**Comparison to Prior Accelerators:** Figure 6 classifies prior accelerators using our algorithm variants taxonomy and compares their 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, $\text{Sync}_{\text{slice}}\text{Sliced}_{\text{rndrbn}}$ is the best variant, as it balances work-efficiency and memory-efficiency.

## VIII. Discussion and Conclusion

| Accelerators | Update Visibl. | Vertex Sched. | Slice Sched. | Update Dirn. |
|---|---|---|---|---|
| Graphcnd. [5] | Sync | Locality | Slice-round | Push |
| Ozdal [7] | Async | Locality | Non-sliced | Pull |
| Chronos [1] | Async | WorkEff | Non-sliced | Pull |
| GraphPls. [10] | Async | Locality | Sliced-loc | Push |
| **PG-SingleAlg (Ours)** | **Async** | **WorkEff** | **Sliced-loc** | **Push** |

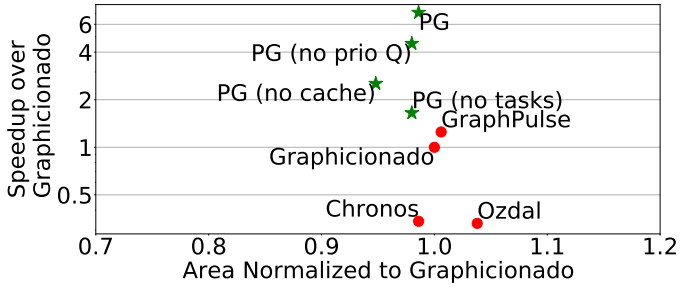


**Fig. 6: Performance Comparison with Prior Accelerators.**

As the scale and scope of graph processing continues to expand, there is an increasing need for robust and high-performance accelerators. Yet, a universal graph-processing accelerator has not yet been found; our results support the viewpoint that this is because many algorithm variants work dramatically better for specific combinations of graph inputs and workloads. Our approach embraces this complexity by creating an accelerator execution model, Taskflow, that flexibly supports important graph algorithm variants.

To understand the value of flexibility in this complex space with much prior art, the crux of our approach was to systematize the codesign process: We created a modular execution model and accelerator which can optionally support hardware features leading to efficient execution of certain algorithm variants (e.g. caches for non-sliced execution, remote tasks/atomics for push-based algorithms, etc.). This framework enabled us to make both strong statements about the effectiveness of various algorithm variants, as well as about the value of flexibility across these codesign dimensions – without being hampered by differences in evaluation (e.g. simulator & preprocessing assumptions). This systematic approach could help understand other important and complex domains (e.g. databases) for pushing the bounds of accelerator capabilities.

For graph processing, we found that flexibility is essential for even a modest range of graph workloads, and we believe this has significance for future research in this area. An aggressive view is that this predicts the decline of narrowly defined graph accelerators, and perhaps accelerators in other irregular domains (e.g. recent sparse tensor work makes similar observations [8,9]). At a minimum, to get the most insight, future evaluations should use baselines with multiple algorithm variants and avoid narrow selections of input types.

Beyond graph processing, our work is the first step towards unifying task parallelism and dataflow acceleration, and this has implications for future programmable accelerators. Up to this point, reconfigurable dataflow processors (e.g. [4,6]) were only suitable for static-parallel workloads; dynamic task parallelism on such architectures would have required centralized coordination and pipeline fill/drain overheads that overwhelm short tasks. Taskflow makes tasks a first-class primitive in the dataflow model: task nodes introduce a breaking-point in the pipelined dataflow to re-order tasks in time (for work-efficiency), or in space (to enable near-data processing). Tasks can also be scheduled in hardware, enabling distributed task management, and efficient task pipelining of even tiny tasks. Overall, the unification of task parallelism and dataflow is a new direction with the potential to broaden the scope of future programmable accelerators.

## Biographies

**Vidushi Dadu** is currently working toward the Ph.D. degree with the Department of Computer Science, University of California Los Angeles. Her research interests include hardware–software codesign and programmable acceleration. Dadu received the B.Tech. degree in Electronics and Communication Engineering from the Indian Institute of Technology Roorkee. She is a student member of IEEE. Contact her at `vidushi.dadu@cs.ucla.edu`.

**Sihao Liu** is currently working toward the Ph.D. degree with the Department of Computer Science, University of California Los Angeles. His research interests include spatial architecture prototyping and design space exploration. Liu received the B.Eng. degree in Electrical Engineering from Xi'an Jiaotong University. He is a student member of IEEE. Contact him at `sihao@cs.ucla.edu`.

**Tony Nowatzki** is an Assistant Professor with the Department of Computer Science, University of California Los Angeles. His research interests include architecture and compiler codesign and novel hardware/software interfaces. Nowatzki received a Ph.D. in Computer Science from the University of Wisconsin-Madison. He is a member of IEEE. Contact him at `tjn@cs.ucla.edu`.

## References

[1] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," ser. ASPLOS '20, 2020.
[2] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *INFOCOM*, 2000.
[3] V. Dadu, S. Liu, and T. Nowatzki, "Polygraph: Exposing the value of flexibility for graph processing accelerators," in *ISCA*, 2021.
[4] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," ser. MICRO '52, 2019.
[5] 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.
[6] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," ser. ISCA '17, 2017.
[7] 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.
[8] S. Pal, A. Amarnath, S. Feng, M. O'Boyle, R. Dreslinski, and C. Dubach, "Sparseadapt: Runtime control for sparse linear algebra on a reconfigurable accelerator," in *MICRO*, ser. MICRO '21, 2021.

[9] E. Qin, R. Garg, A. Bambhaniya, M. Pellauer, A. Parashar, S. Rajamanickam, C. Hao, and T. Krishna, "Enabling flexibility for sparse tensor acceleration via heterogeneity," *arXiv preprint arXiv:2201.08916*, 2022.

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

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

[12] 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.