VLSI architecture, synthesis & technology

UCLA

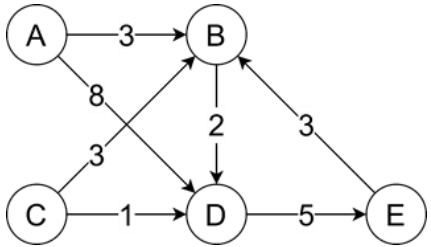
Accelerating SSSP for Power-Law Graphs

Yuze Chi, Licheng Guo, Jason Cong

{chiyuze,lcguo,cong}@cs.ucla.edu

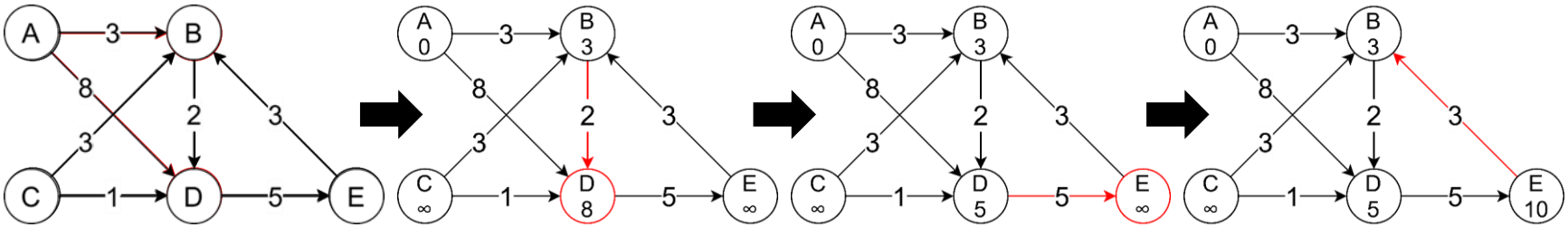
Single-Source Shortest Path (SSSP)

- Widely used in many applications
  - Road navigation
  - Telecom network routing
  - Neural image reconstruction
  - Social network analysis
- For a given positive-weighted, directed graph $G = (V, E)$
  - An undirected edge → two directed edges
  - Given a source vertex $u \in V$
- Find the shortest-path tree from u, including
  - The *shortest distance* from u to each vertex
  - The *parent* of each vertex



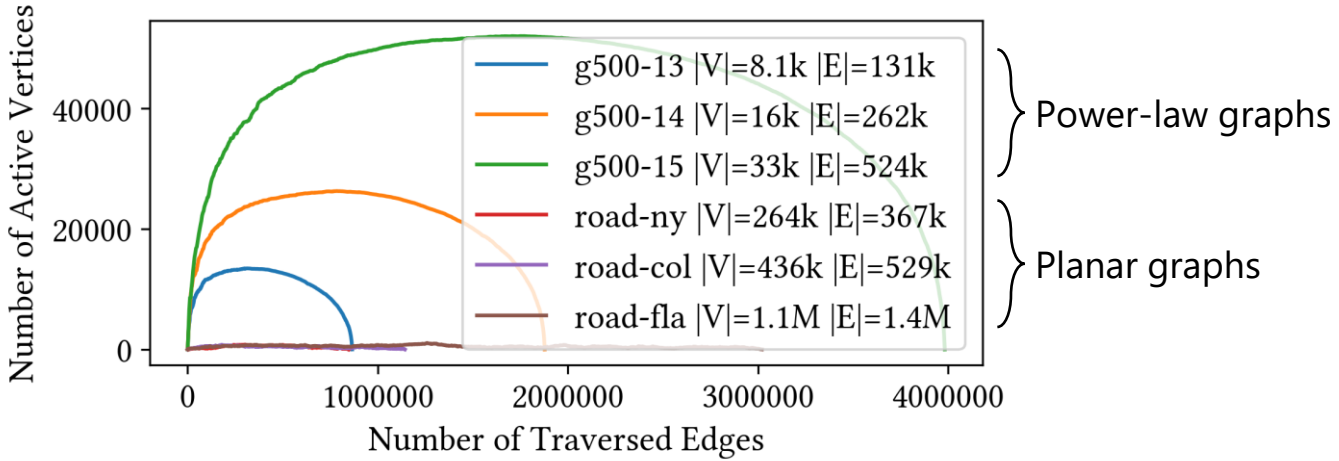
Dijkstra's Algorithm (w/ Priority Queue)

- Relax edges only when necessary
- Store active vertices in a priority queue
- Only relax edges from the shortest known distance
- No redundant work



SSSP for Power-Law Graphs is Challenging

- Planar graphs: *small* frontier, *does not* scale with graph size
- Power-law graphs: *large* frontier, *does* scale with graph size
  - One of the kernels of Graph 500 benchmarks



SSSP for Power-Law Graphs is Challenging

- High-performance SSSP for power-law graphs requires
  - *High-throughput* and *high-capacity* **priority queue** for *high work-efficiency*
  - Fast **memory system** with high random-access throughput for *fast edge traversal*

System	Lang-uage	Algorithm	Work-efficient?	Power-law?	Priority queue?	Vertex cache?	Maximum MTEPS
Chronos[ASPLOS'20]	RTL	Dijkstra's Variant	Yes	No	P-heap	App.-agnostic	360
GraphLily[ICCAD'21]	HLS	Bellman-Ford	No	Yes	No	Scratchpad	<232
HitGraph[TPDS'19]	RTL	Bellman-Ford	No	Yes	No	Scratchpad	46.9
Lei et. al[TCAS-II'16]	RTL	Dijkstra's Variant	Yes	No	ExSAPQ	No	9.2
Takei et. al[PDPTA'15]	RTL	Dijkstra's	Yes	No	No	On-chip only	0.4
ThunderGP[FPGA'21]	HLS	Bellman-Ford	No	Yes	No	Scratchpad	<122
SPLAG[FPGA'22]	**HLS**	**Dijkstra's Variant**	**Yes**	**Yes**	**CGPQ**	**CVC**	**763**

Solution: Relaxing the Priority Definition

- Strict priority
  - Distance from root → fine-grained (e.g., 32 bit `float` distance)
  - Binary heap/pipelined heap/systolic priority queue/etc.
  - Hard to scale to large size with a high throughput

- Relaxed priority
  - Distance from root/Δ → coarse-grained (e.g., 6 bit `uint` bucket)
  - High throughput: concurrent push & pop
  - Scalable to large size with high throughput

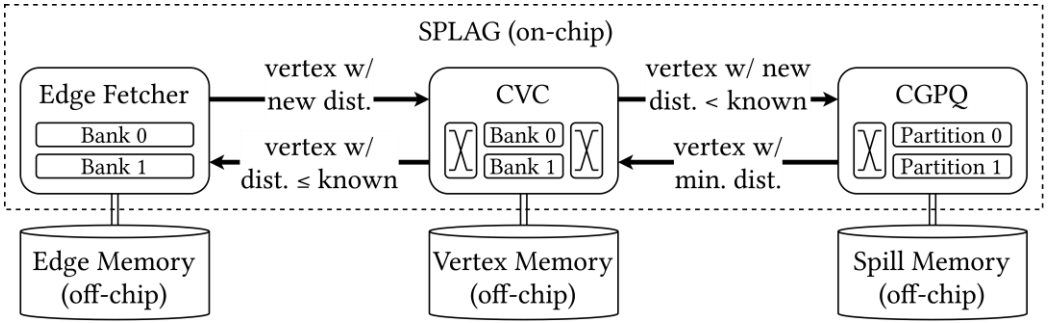
SPLAG Algorithm

- Require: A graph $G = (V, E)$ and $root \in V$
- Ensure: $vertices$ represent the shortest-path tree from $root$

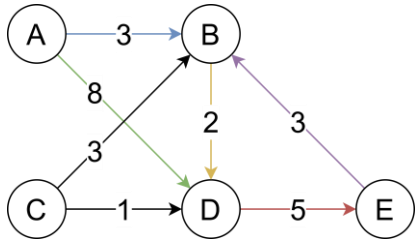
1. $vertices = [\{dist = \infty, parent = \text{null}\}, \cdots]$
2. $vertices[root] = [\{dist = 0, parent = root\}]$
3. $queue = [\{id = root, dist = 0, parent = root\}]$
4. **while not** $queue$.empty() **in parallel do**
5. $u = queue$.pop() ◁ CGPQ
6. **if** $u.dist \leq vertices[u.id].dist$ **then** ◁ CVC
7. **for all** $e = u.id \rightarrow vid \in E$ **in parallel do** ◁ Edge Fetcher
8. **if** $vid \neq u.parent$ **then** ◁ Edge Fetcher
9. $d = u.dist + e.weight$ ◁ Edge Fetcher
10. **if** $d < vertices[vid].dist$ **then** ◁ CVC
11. $vertices[vid] = \{dist = d, parent = u.id\}$ ◁ CVC
12. $queue$.push($\{id = vid, dist = d, parent = u.id\}$) ◁ CGPQ

SPLAG Architecture

- Each component is internally partitioned & parallelized
- Multi-stage network used for inter-partition communication

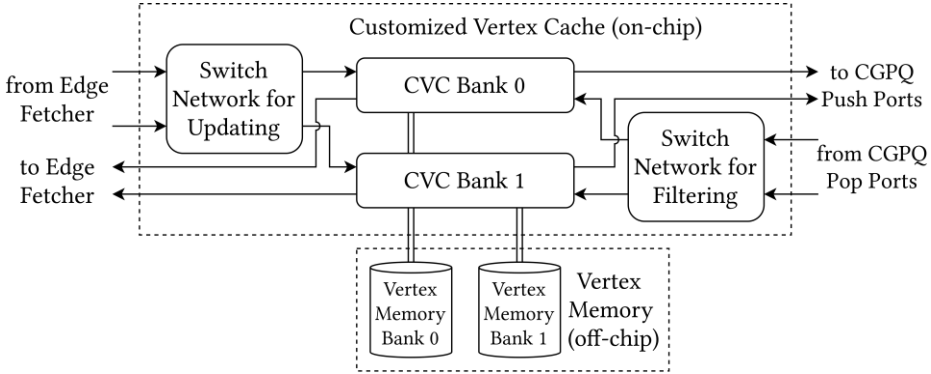


(8 HBM channels) (16 HBM channels) (8 HBM channels)



Customized Vertex Cache (CVC)

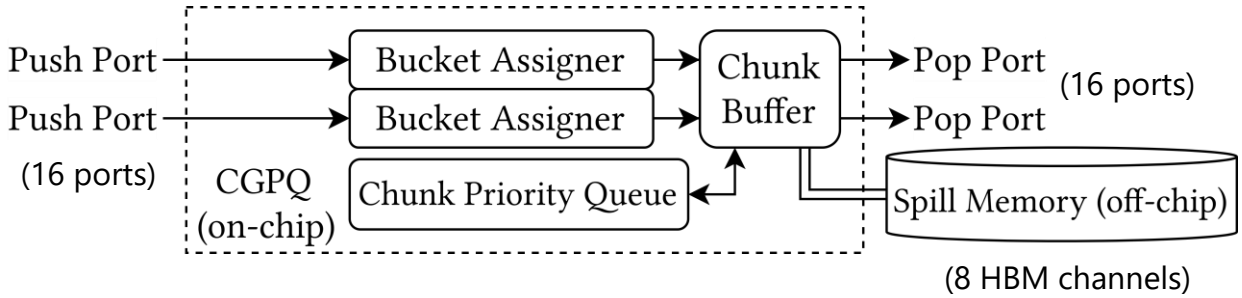
- Direct-mapped, write-back
- Fully pipelined
  - Hiding off-chip memory latency
  - Leverages `tapa::async_mmap`
- *Application-specific* operations
  - **Updating** vertices atomically from the edge fetcher
  - **Filtering** vertices from CGPQ to the edge fetcher
  - Both operations discard redundant relaxation



(16 banks in practice)

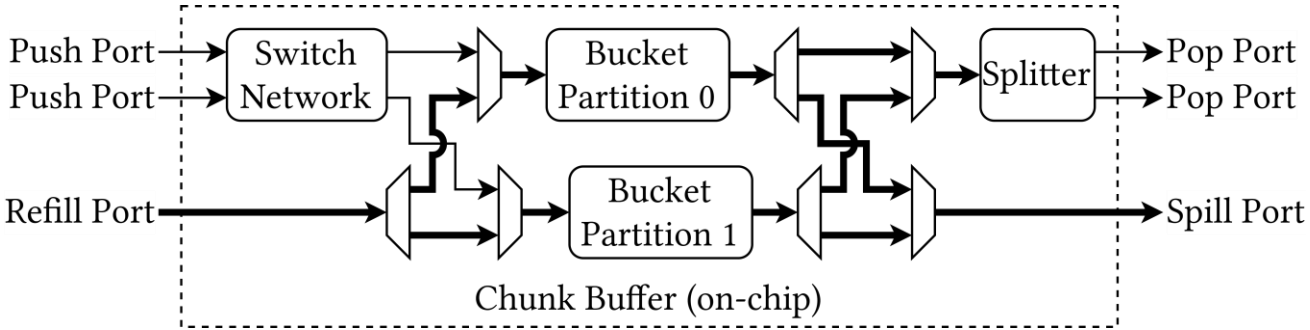
Coarse-Grained Priority Queue (CGPQ)

- Inspired by “deque”
  - Assign bucket ID based on distance
  - Divide each bucket into chunks
  - Buffer 1 chunk per bucket
  - Spill excessive chunks off-chip
  - Refill chunk with shortest distance



CGPQ Chunk Buffer Design

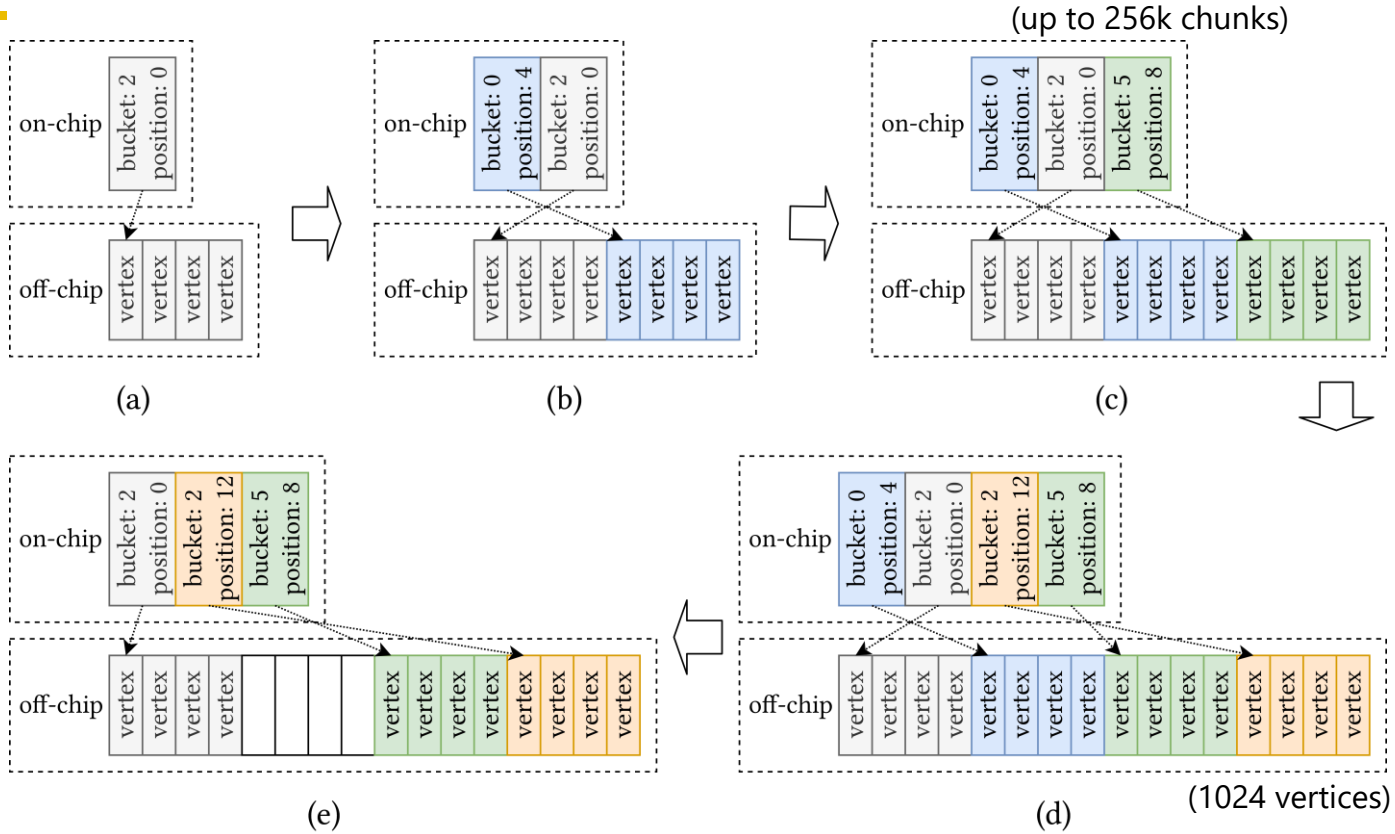
- Highly concurrent design for *high throughput*
  - Inter-bucket & intra-bucket parallelism
- Support spilling & refilling for *high capacity*
  - Spilling & refilling in unit of chunks



CGPQ Spilling & Refilling

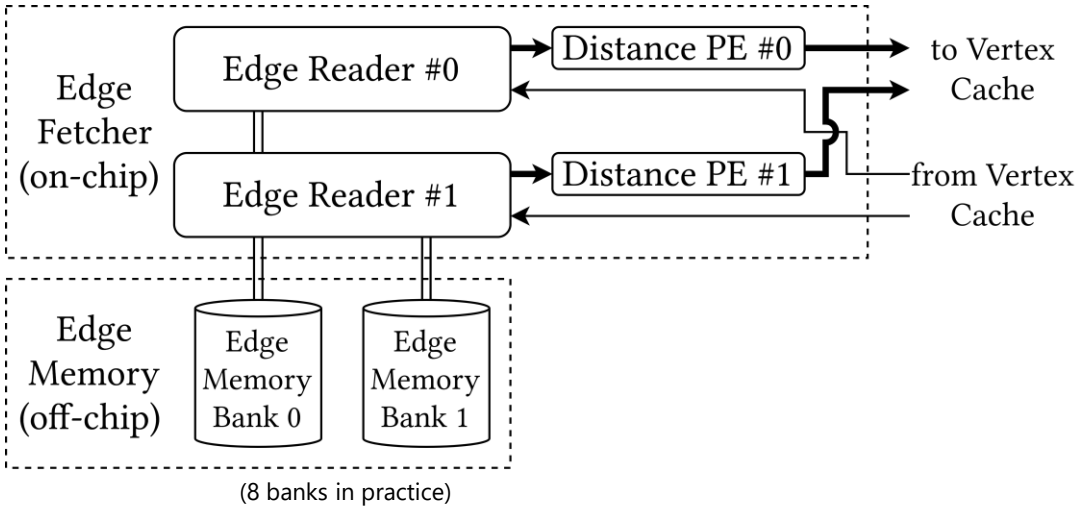
- The chunk buffer keeps track of the size of each buffered chunk
- A chunk spills into DRAM when it is almost full
  - Location in DRAM and priority of spilled chunk are stored in an on-chip chunk priority queue (CPQ)
- A spilled chunk refills into chunk buffer when
  - It is the highest-priority chunk, and it is almost empty
  - The on-chip CPQ finds the highest-priority chunk

CGPQ Spilling & Refilling Example



Edge Fetcher

- Handles neighbor edge traversal
- Computes a new tentative distance of each neighbor

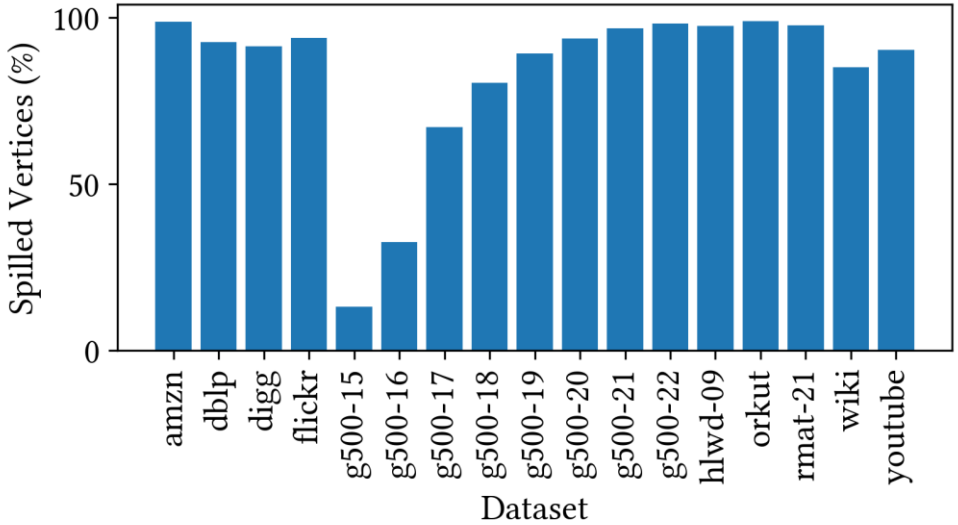


Evaluation: Datasets

Dataset	#Vertex	#Edge	Maximum Degree	Average Degree	Source
amzn	2.1M	6M	12k	3	Amazon product ratings
dblp	0.5M	15M	3k	28	DBLP paper coauthors
digg	0.9M	4M	31k	4	Users from digg.com
flickr	2.3M	33M	34k	14	Flicker users
g500-N	2^N	2^{N+4}	$2^{0.6N+5}$	16	Graph 500 datasets
hlwd-09	1.1M	58M	12k	50	Actor collaboration
orkut	3.0M	106M	28k	36	Orkut social network
rmat-21	2.1M	91M	214k	44	A Kronecker graph
wiki	0.3M	3M	3k	11	Wiki article-word graph
youtube	3.2M	12M	130k	4	YouTube users

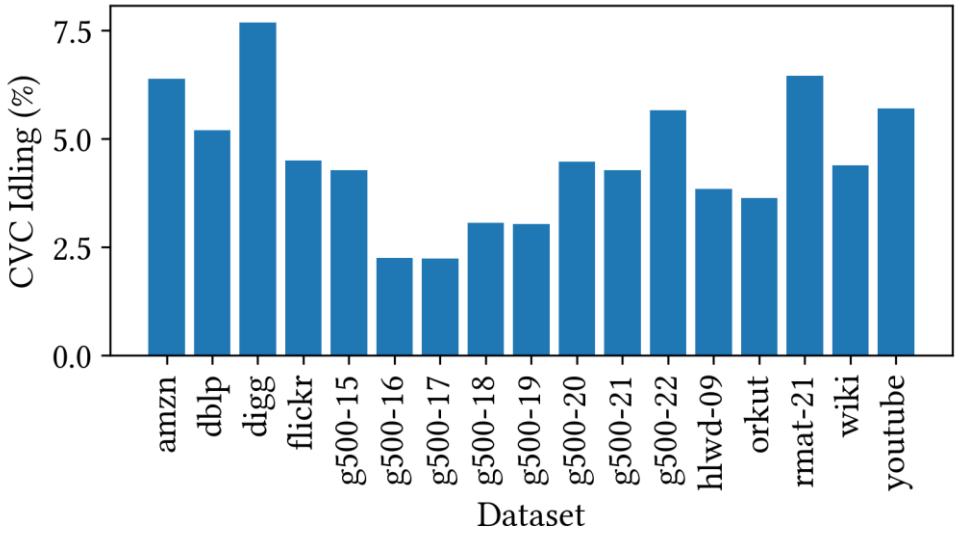
Evaluation: CGPQ Capacity

- Spilling is effective
- Spilling is scalable
- **High-capacity** PQ ✔



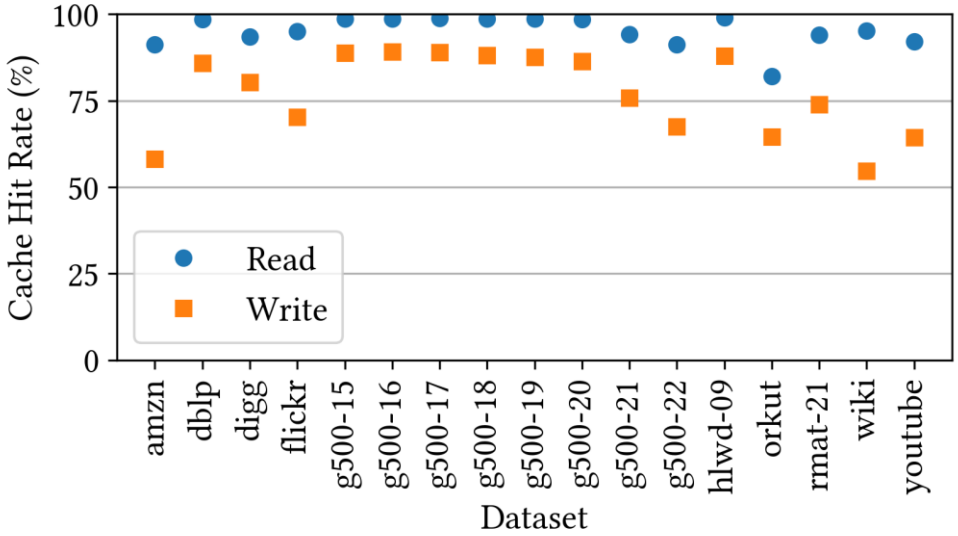
Evaluation: CGPQ Throughput

- CVC idling <8%
  - May be caused by
    - Empty CGPQ
    - Pop throughput too slow
- CGPQ never stalls CVC
- **High-throughput** PQ ✔



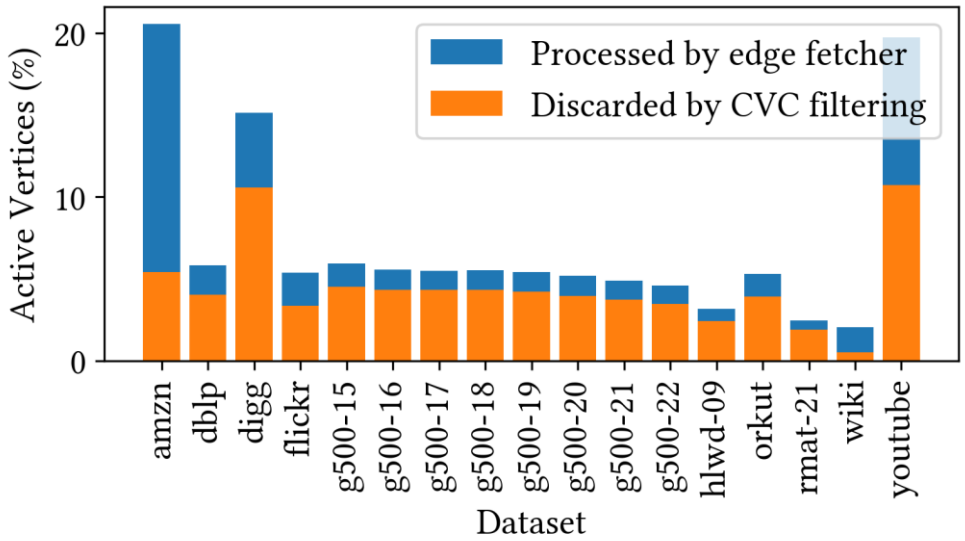
Evaluation: CVC Hit Rate

- Read hit rate
  - $100\% - \frac{\#\text{off-chip reads}}{\#\text{reads}}$
  - >80%
- Write hit rate
  - $100\% - \frac{\#\text{off-chip writes}}{\#\text{writes}}$
  - >50%
- **Fast memory system** ✔



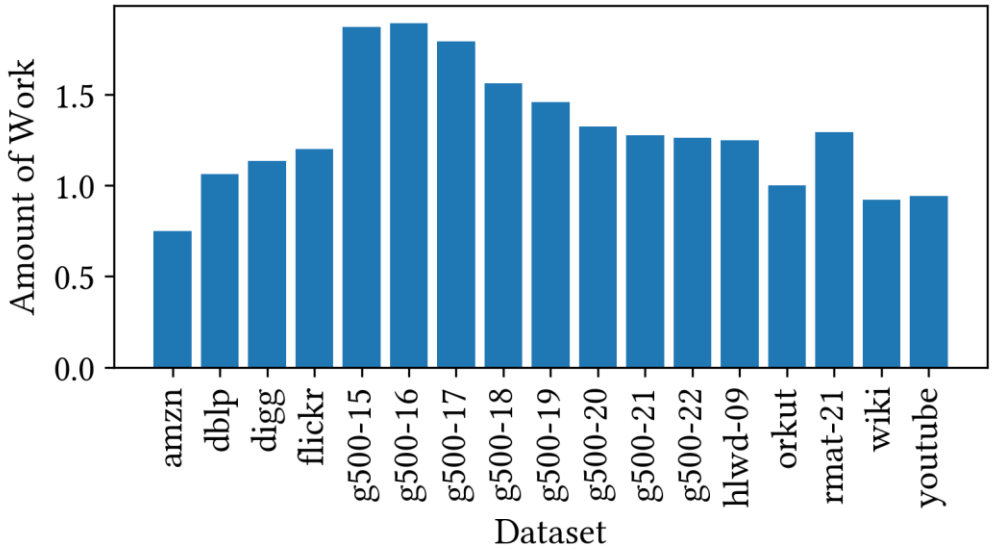
Evaluation: Edge Traversal Breakdown

- Each traversed edge may
  - Be discarded
    - By CVC updating (not shown)
  - Be pushed to CGPQ
    - Generates active vertices 👉
- Active vertices may
  - Be processed by
    - The edge fetcher (bar)
  - Be discarded by
    - CVC filtering (bar)



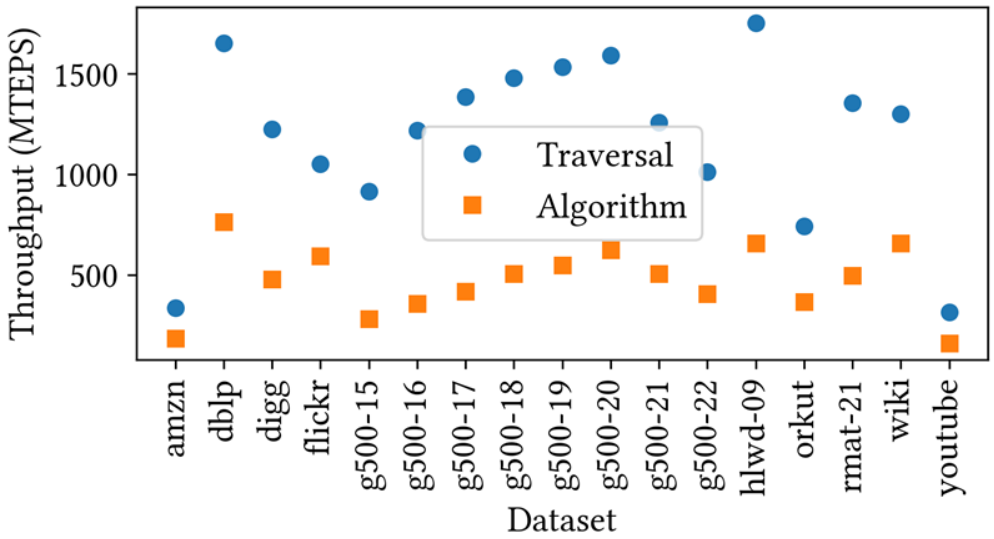
Evaluation: Work Efficiency

- Amount of work
  - $\frac{\#\text{edges traversed}}{\#\textit{directed}\text{ edges in connected component}}$
- **Work-efficient** ✔



Evaluation: Throughput

- Traversal throughput
  - $\frac{\#\text{edges traversed}}{\text{execution time}}$
- Algorithm throughput
  - $\frac{\#\textit{undirected}\text{ edges in connected component}}{\text{execution time}}$
  - Metric used by Graph 500



Evaluation: Comparison

Dataset	System	Hardware	Algorithm	Throughput (MTEPS)		SPLAG's Speedup
				Traversal	Algorithm	
hlwd-09	Galois[PLDI'07]	Xeon 6244 CPU	Δ-Stepping	1229	211	**2.6×**
	ADDS[PPoPP'21]	A100 40G GPU	ADDS	31242	1455	0.4×
	GraphLily[ICCAD'21]	U280 FPGA	Bellman-Ford	4670	<232	**>2.3×**
	SPLAG[FPGA'22]	U280 FPGA	SPLAG	1744	**543**	1×
	ThunderGP[FPGA'21]	U250 FPGA	Bellman-Ford	2454	<122	**>2.6×**
	SPLAG[FPGA'22]	U250 FPGA	SPLAG	756	**315**	1×
rmat-21	Galois[PLDI'07]	Xeon 6244 CPU	Δ-Stepping	930	254	**1.9×**
	ADDS[PPoPP'21]	A100 40G GPU	ADDS	15878	530	0.9×
	GraphLily[ICCAD'21]	U280 FPGA	Bellman-Ford	2823	<195	**>2.5×**
	SPLAG[FPGA'22]	U280 FPGA	SPLAG	1354	**494**	1×
	HitGraph[FPGA'21]	VU5P FPGA	Bellman-Ford	2152	46.9	**4.9×**
	SPLAG[FPGA'22]	VU5P FPGA	SPLAG	533	**228**	1×
g500-21	**SPLAG**[FPGA'22]	U280 FPGA	SPLAG	1257	**504**	1×

In a Nutshell

- SPLAG is
  - Performant & energy-efficient
  - Portable & open-source (https://github.com/UCLA-VAST/splag)
- Thanks to
  - CGPQ & CVC
  - TAPA (https://github.com/UCLA-VAST/tapa)
- Feedback & suggestions are appreciated

References

- Δ-stepping: a parallelizable shortest path algorithm, Meyer & Sanders, in Journal of Algorithms, 2003.
- [Galois]: Optimistic Parallelism Requires Abstractions, Kulkarni et al., in PLDI, 2007.
- [Takei et al.]: Evaluation of an FPGA-Based Shortest-Path-Search Accelerator, Takei et al., In PDPTA, 2015.
- [Lei et al.]: An FPGA Implementation for Solving the Large Single-Source-Shortest-Path Problem, Lei et al., in TCAS-II, 2016.
- HitGraph: High-Throughput Graph Processing Framework on FPGA, Zhou et al., in TPDS, 2019.
- Chronos: Efficient Speculative Parallelism for Accelerators, Abeydeera & Sanchez, in ASPLOS, 2020.
- ThunderGP: HLS-Based Graph Processing Framework on FPGAs, Chen et al., in FPGA, 2021.
- [ADDS]: A Fast Work-Efficient SSSP Algorithm for GPUs, Wang et al., in PPoPP, 2021.
- GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs, Hu et al., in ICCAD, 2021.

Acknowledgment

This work is partially supported by the NSF RTML program (CCF1937599), NIH Brain Initiative (U01MH117079), the Xilinx Adaptive Compute Clusters (XACC) program, CRISP, one of six JUMP centers, and support of the CDSC industrial partners.