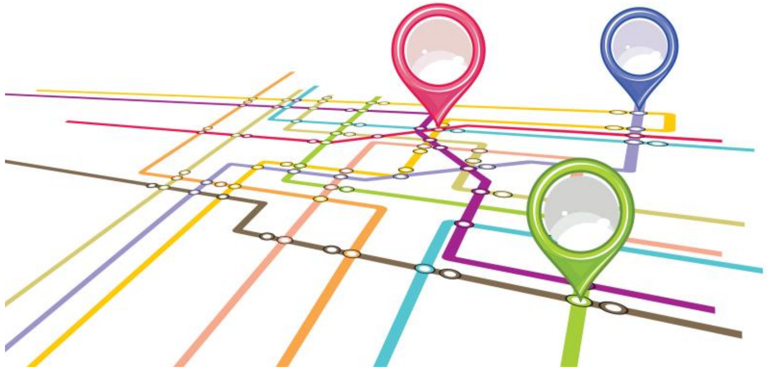
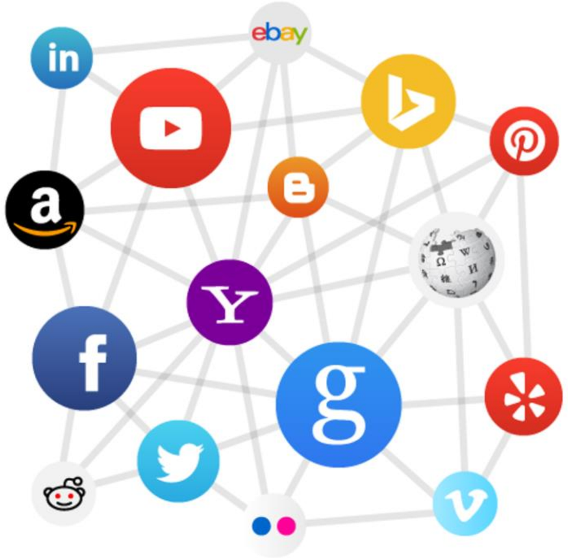
GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs

Yuwei Hu, Yixiao Du, Ecenur Ustun, Zhiru Zhang

Cornell University

Graph Processing Is Ubiquitous

Social network	Railway	Internet
		
Breadth-first search (BFS)	Single-source shortest path (SSSP)	PageRank
Recommend 2-hop neighbors as new friends	Navigation	Search engines

Graph Processing on FPGAs

Advantages of using FPGAs:

- Exploit the fine-grained parallelism in graph processing by customizing the memory hierarchy and compute engines
- Consume less power than CPUs and GPUs

Limitations of prior works (e.g., GraphGen [1], ForeGraph [2], HitGraph [3], ThunderGP [4]):

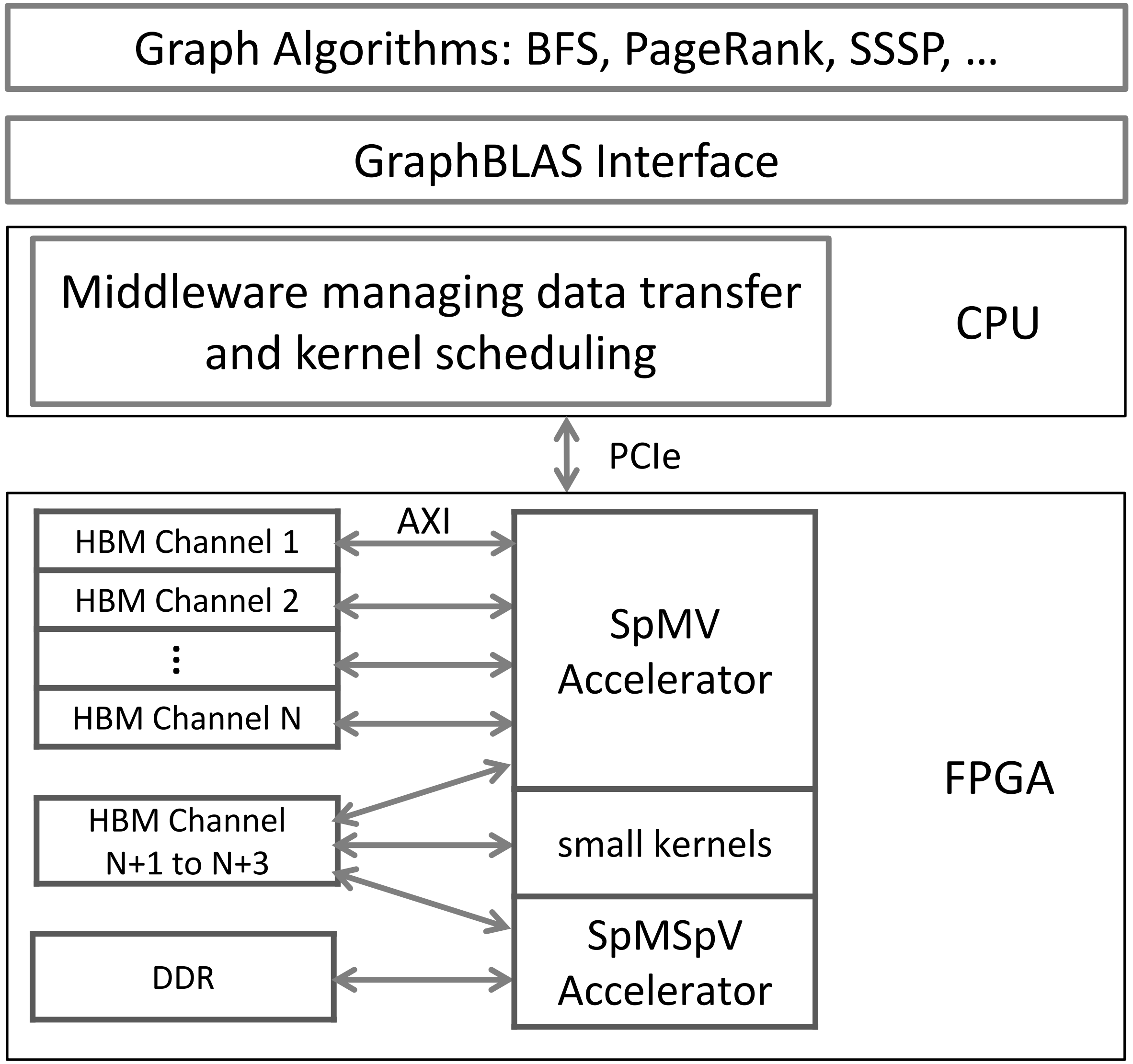
- Require generating/loading a separate bitstream for each graph algorithm
  - Generating a bitstream takes hours or days
  - The cost of switching bitstreams at run time is high
- Target DDR-equipped FPGAs, which have a lower bandwidth than GPUs
  - Graph processing is bandwidth bound

[1] Eriko Nurvitadhi, et al. "An fpga framework for vertex-centric graph computation." FCCM 2014
[2] Guohao Dai, et al. "ForeGraph: Exploring large-scale graph processing on multi-fpga architecture." FPGA 2017
[3] Shijie Zhou, et al. "HitGraph: High-throughput graph processing framework on fpga." TPDS 2019
[4] Xinyu Chen, et al. "ThunderGP: HLS-based graph processing framework on fpgas." FPGA 2021

GraphLily: A Graph Linear Algebra Overlay on HBM-Equipped FPGAs

Contributions:

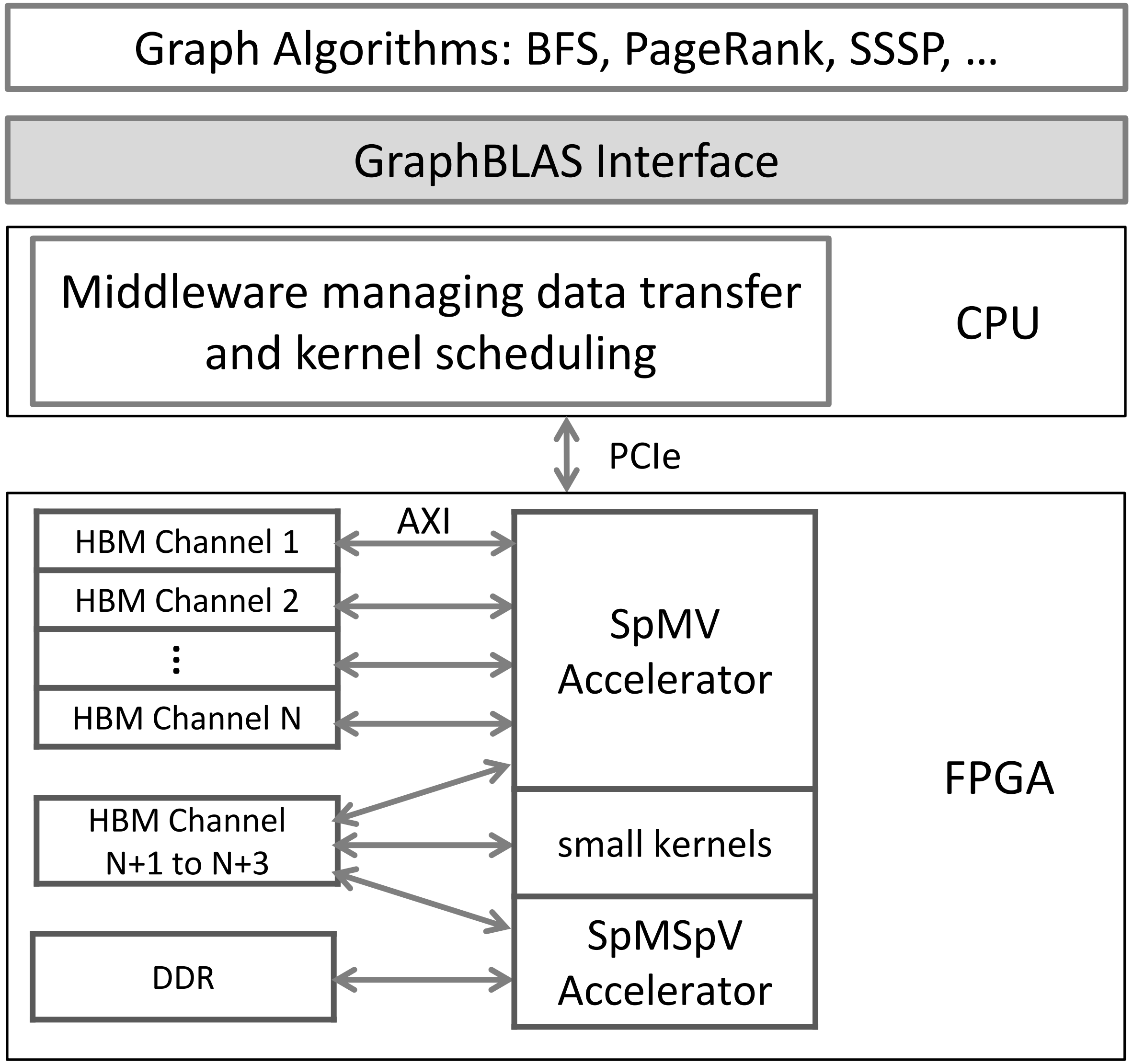
1. The first FPGA overlay for graph processing
2. Effectively utilizing HBM bandwidth by co-designing the data layout and the accelerator architecture
3. Easily porting graph algorithms from CPUs/GPUs to FPGAs with a middleware



GraphLily: A Graph Linear Algebra Overlay on HBM-Equipped FPGAs

Contributions:

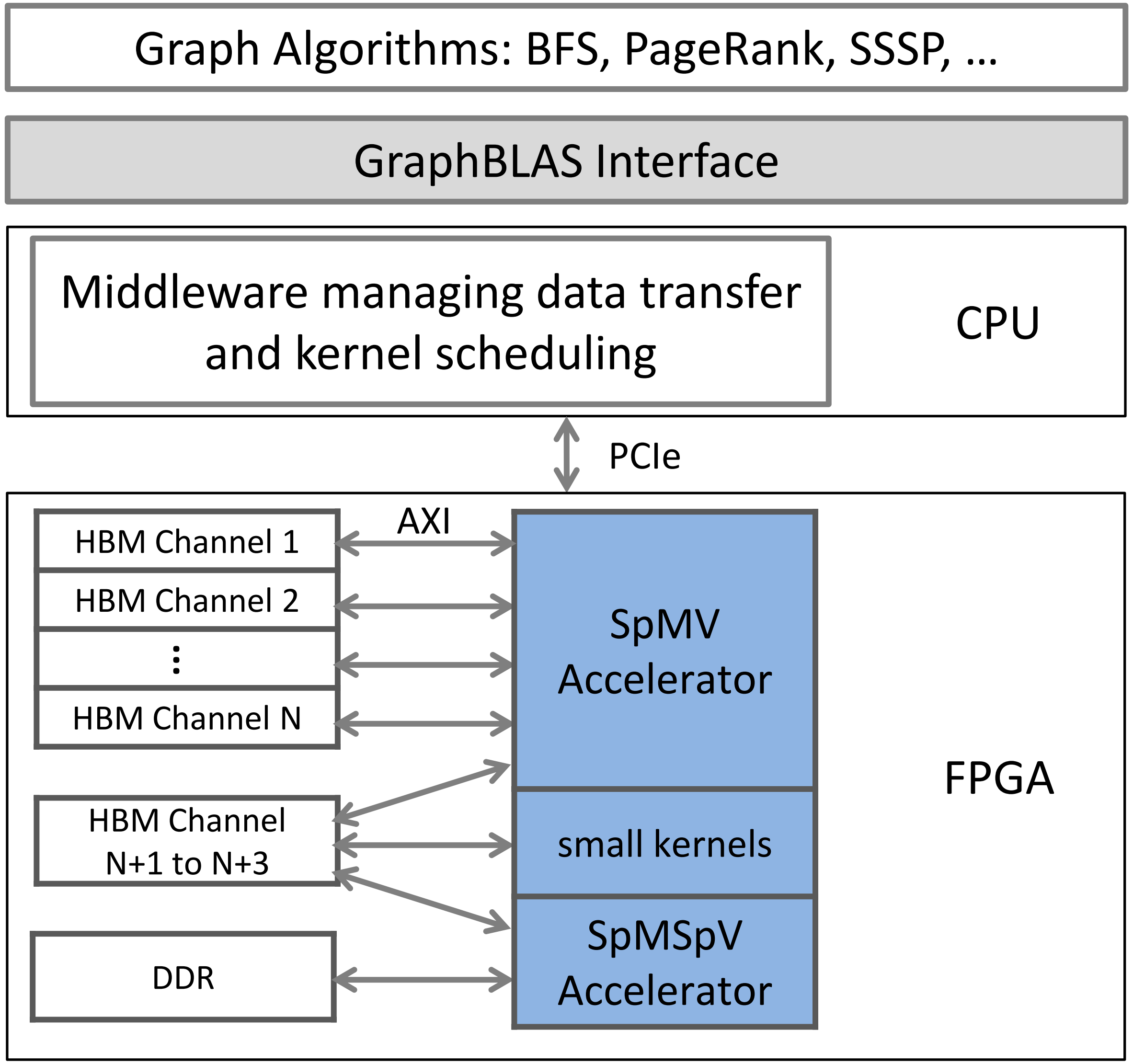
1. The first FPGA overlay for graph processing
2. Effectively utilizing HBM bandwidth by co-designing the data layout and the accelerator architecture
3. Easily porting graph algorithms from CPUs/GPUs to FPGAs with a middleware



GraphLily: A Graph Linear Algebra Overlay on HBM-Equipped FPGAs

Contributions:

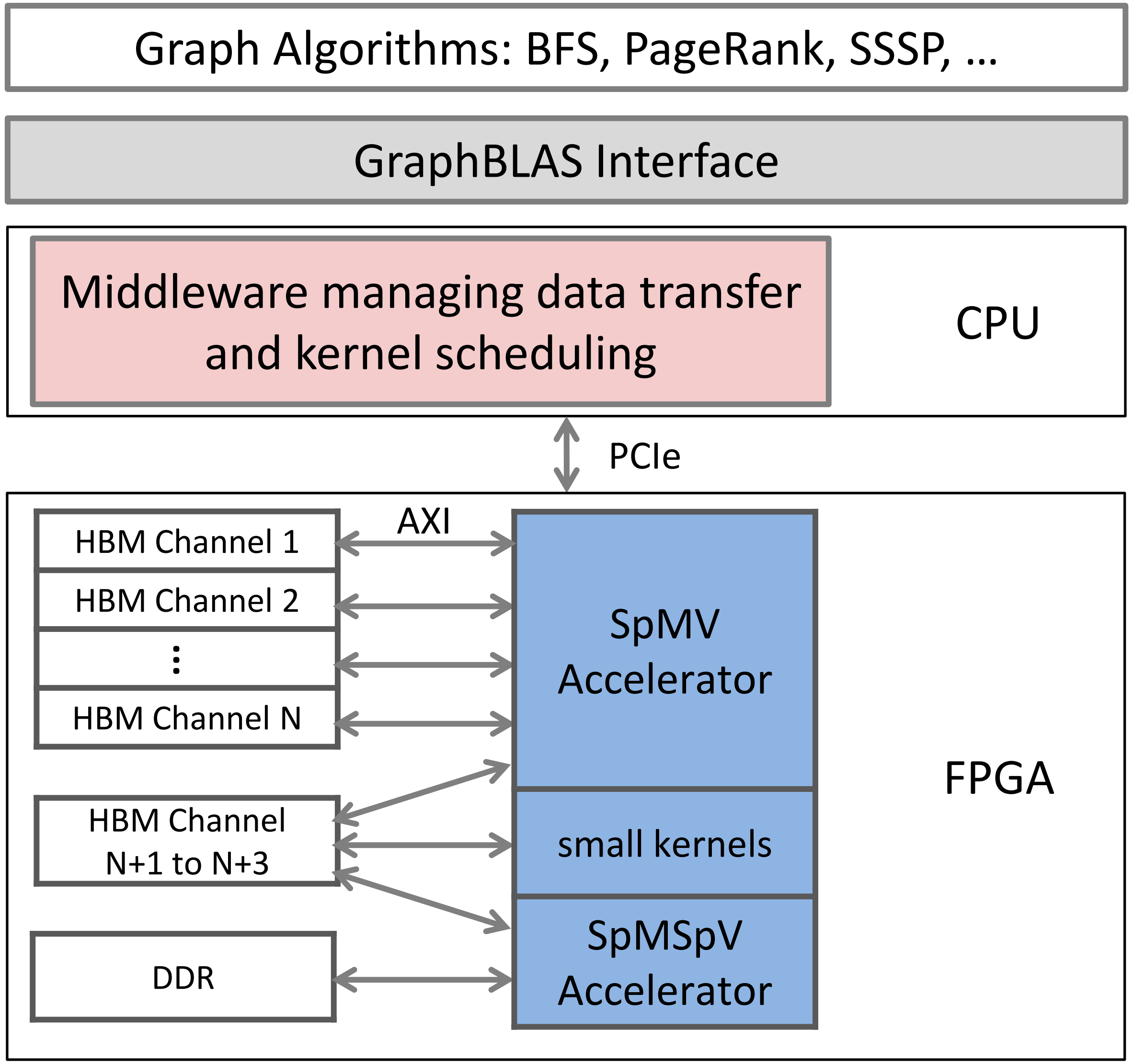
1. The first FPGA overlay for graph processing
2. Effectively utilizing HBM bandwidth by co-designing the data layout and the accelerator architecture
3. Easily porting graph algorithms from CPUs/GPUs to FPGAs with a middleware



GraphLily: A Graph Linear Algebra Overlay on HBM-Equipped FPGAs

Contributions:

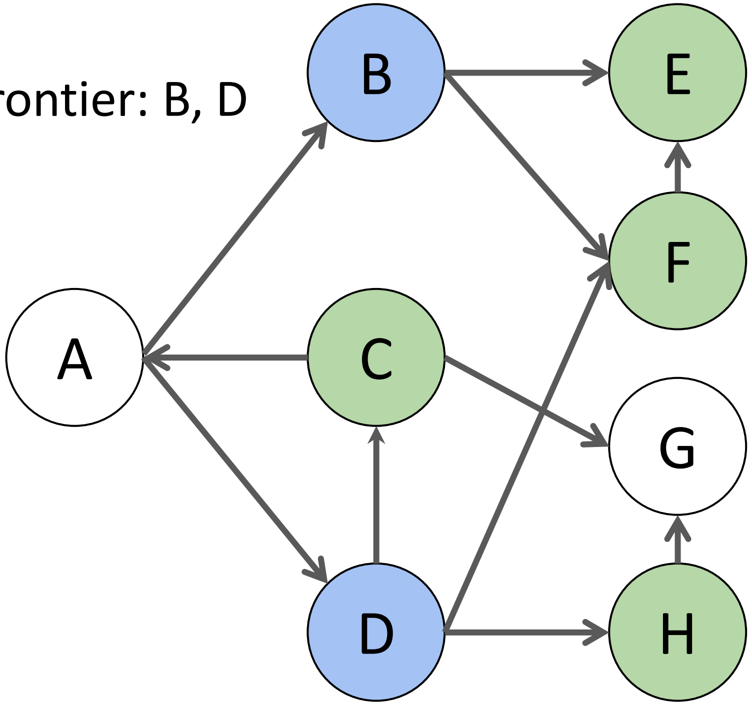
1. The first FPGA overlay for graph processing
2. Effectively utilizing HBM bandwidth by co-designing the data layout and the accelerator architecture
3. Easily porting graph algorithms from CPUs/GPUs to FPGAs with a middleware



Sparse Linear Algebra Formulation of Graph Algorithms

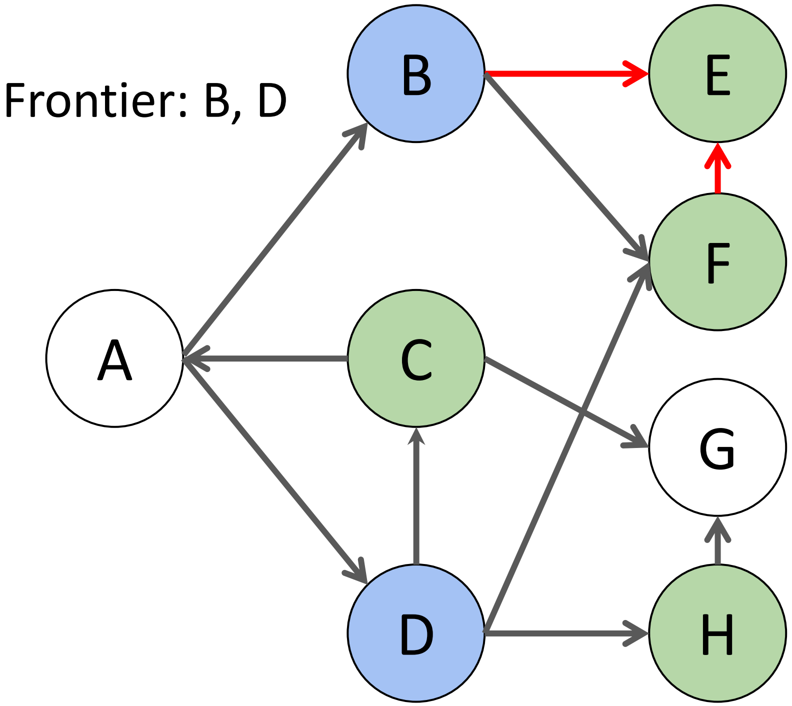
Graph View

Frontier: B, D



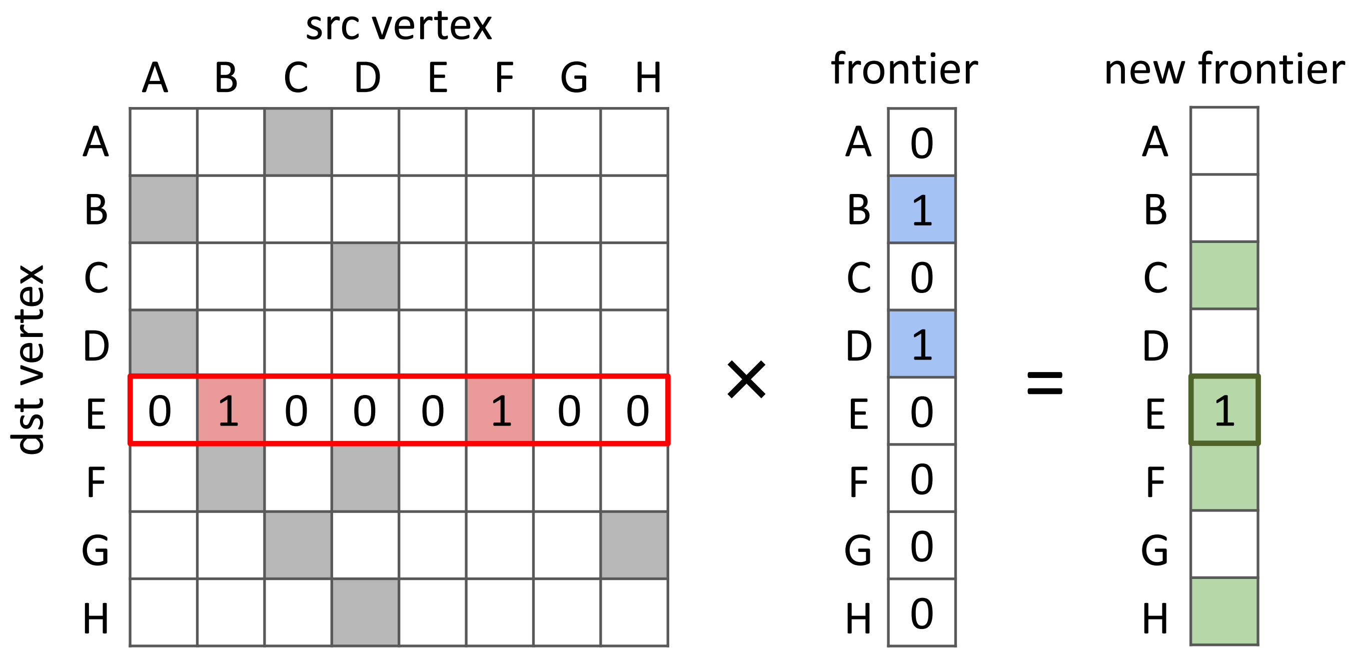
Sparse Linear Algebra Formulation of Graph Algorithms

Graph View



Pull-based graph traversal

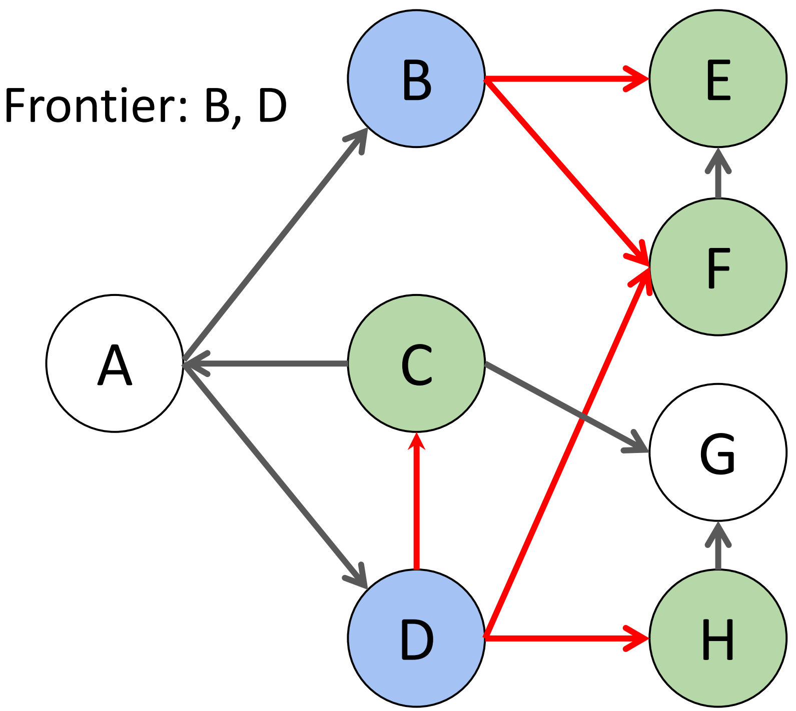
Matrix View



SpMV (sparse-matrix dense-vector multiplication)

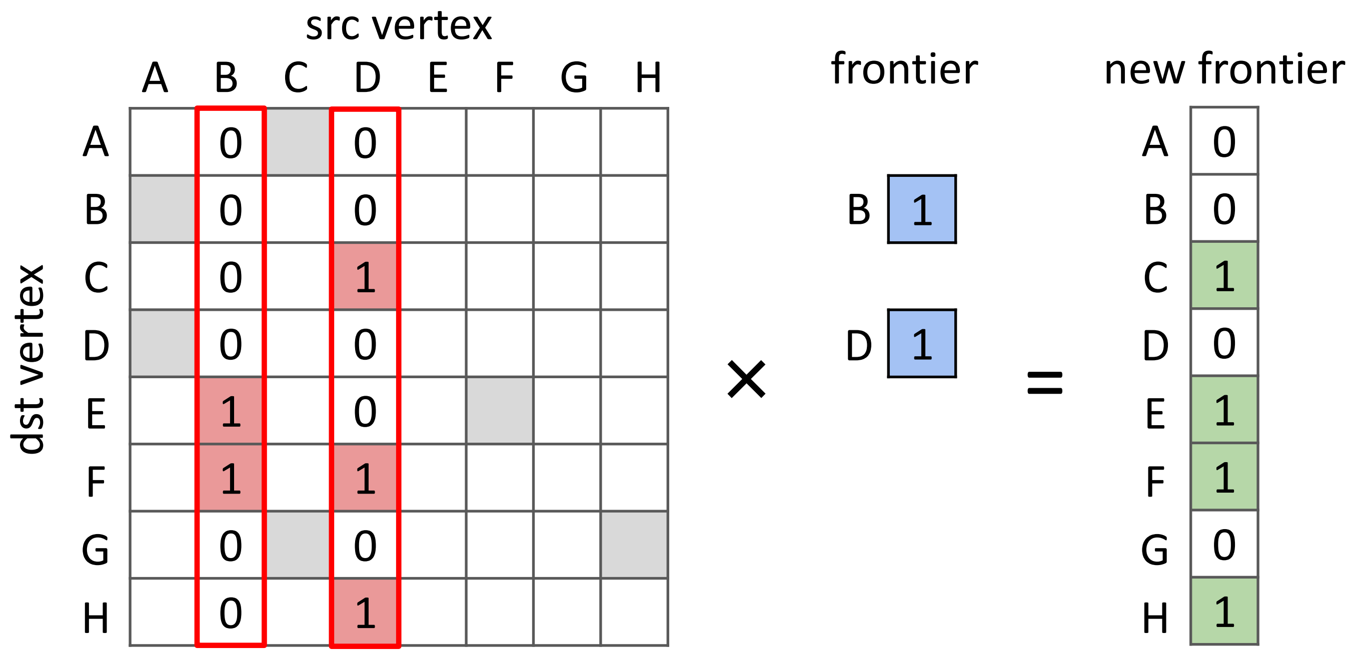
Sparse Linear Algebra Formulation of Graph Algorithms

Graph View



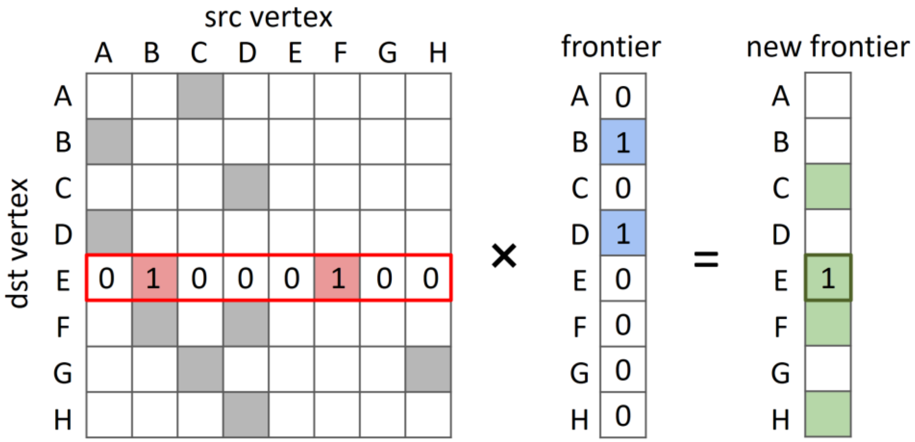
Push-based graph traversal

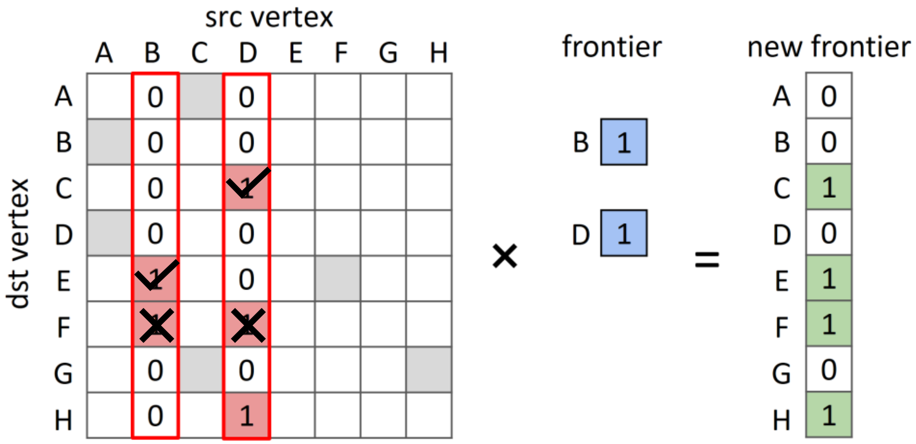
Matrix View



SpMSpV (sparse-matrix sparse-vector multiplication)

SpMV vs. SpMSpV





SpMV:

- More work
- Sequential memory accesses
- Easy to parallelize

SpMSpV:

- Less work
- Random memory accesses
- Hard to parallelize due to contention on updating the output

Heuristic:

- Use SpMSpV when the frontier is small (usually in the first few iterations), switch to SpMV when the frontier is large

GraphBLAS Programming Interface

- Standard building blocks for graph algorithms in the language of sparse linear algebra
- Express a rich set of graph algorithms by generalizing SpMV/SpMSpV:
  - Customizable binary operators and reduction operators, modeled as semirings
  - An optional mask vector (in BFS, the mask vector avoids visiting a vertex twice)

	Binary op	Reduction op	Application
Arithmetic semiring	mul	add	PageRank
Boolean semiring	logical and	logical or	BFS
Tropical semiring	add	min	SSSP

- One API specification, many implementations on CPUs [1][2] and GPUs [3]
  - **GraphLily is the first work that supports GraphBLAS on FPGAs**

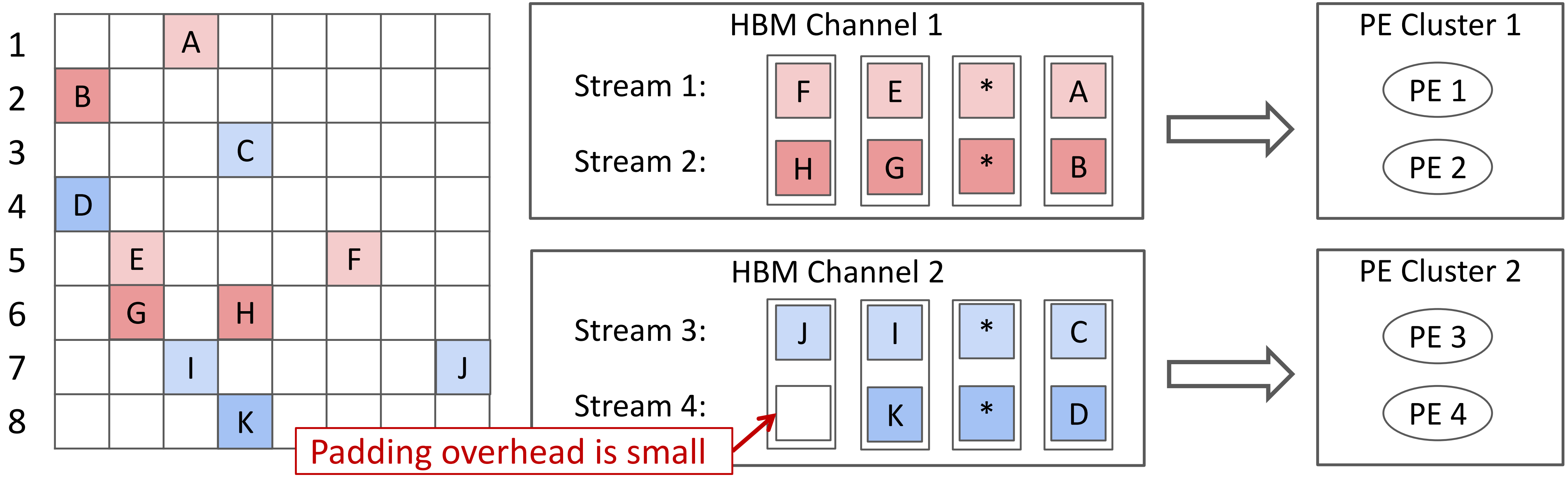
[1] SuiteSparse: https://github.com/DrTimothyAldenDavis/SuiteSparse
[2] Graphblas template library: https://github.com/cmu-sei/gbtl
[3] GraphBLAST: https://github.com/gunrock/graphblast

SpMV Sparse Matrix Format

Saturating the bandwidth of HBM requires:

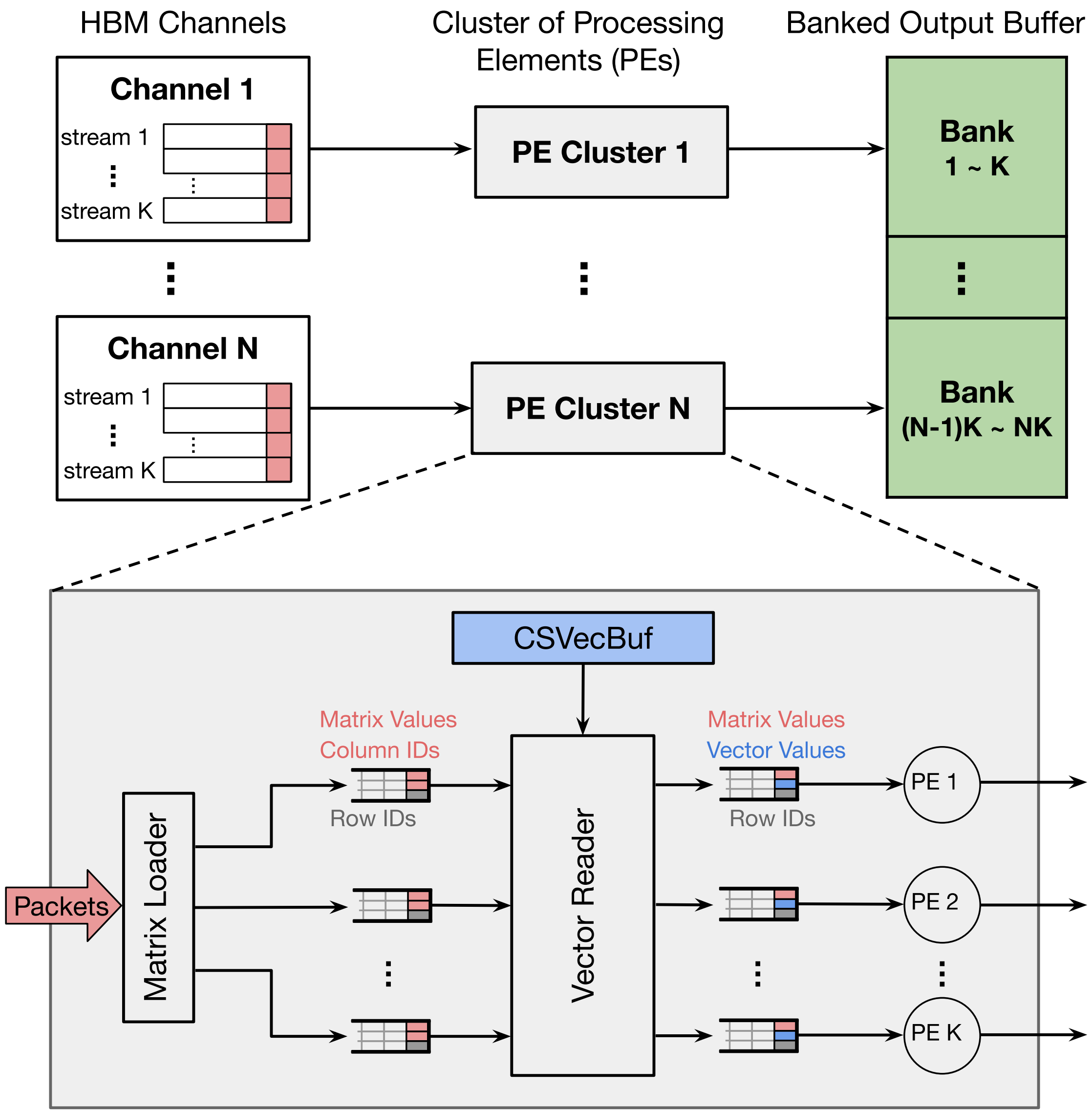
- Vectorized, streaming accesses to each HBM channel
- Concurrent accesses to multiple HBM channels

We propose Cyclic Packed Streams of Rows (CPSR)

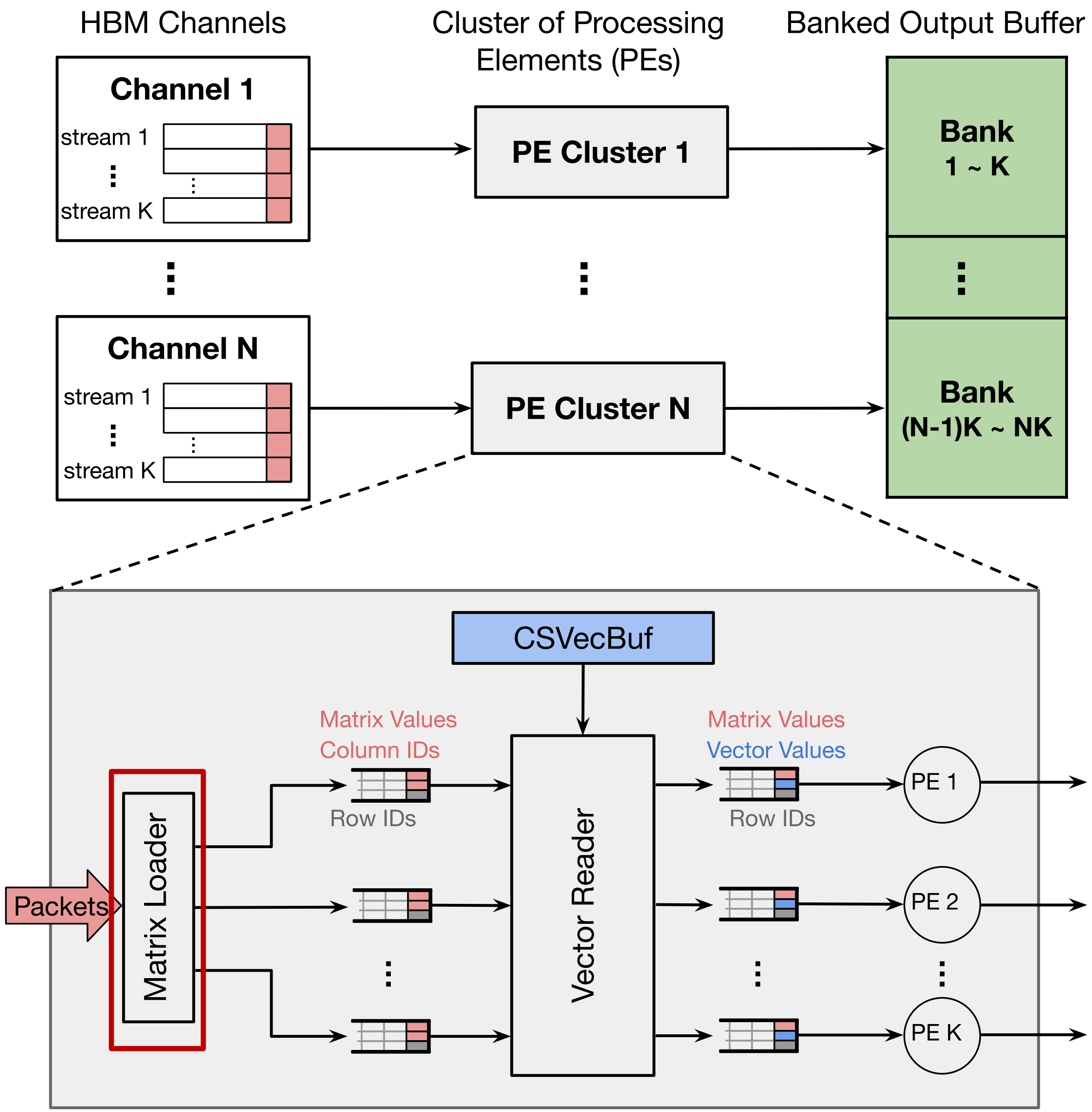


Assume one HBM channel delivers 128 bits per access, one element requires 32 bits for the value and 32 bits for the column index, then the vectorization factor should be 128 / (32 + 32) = 2

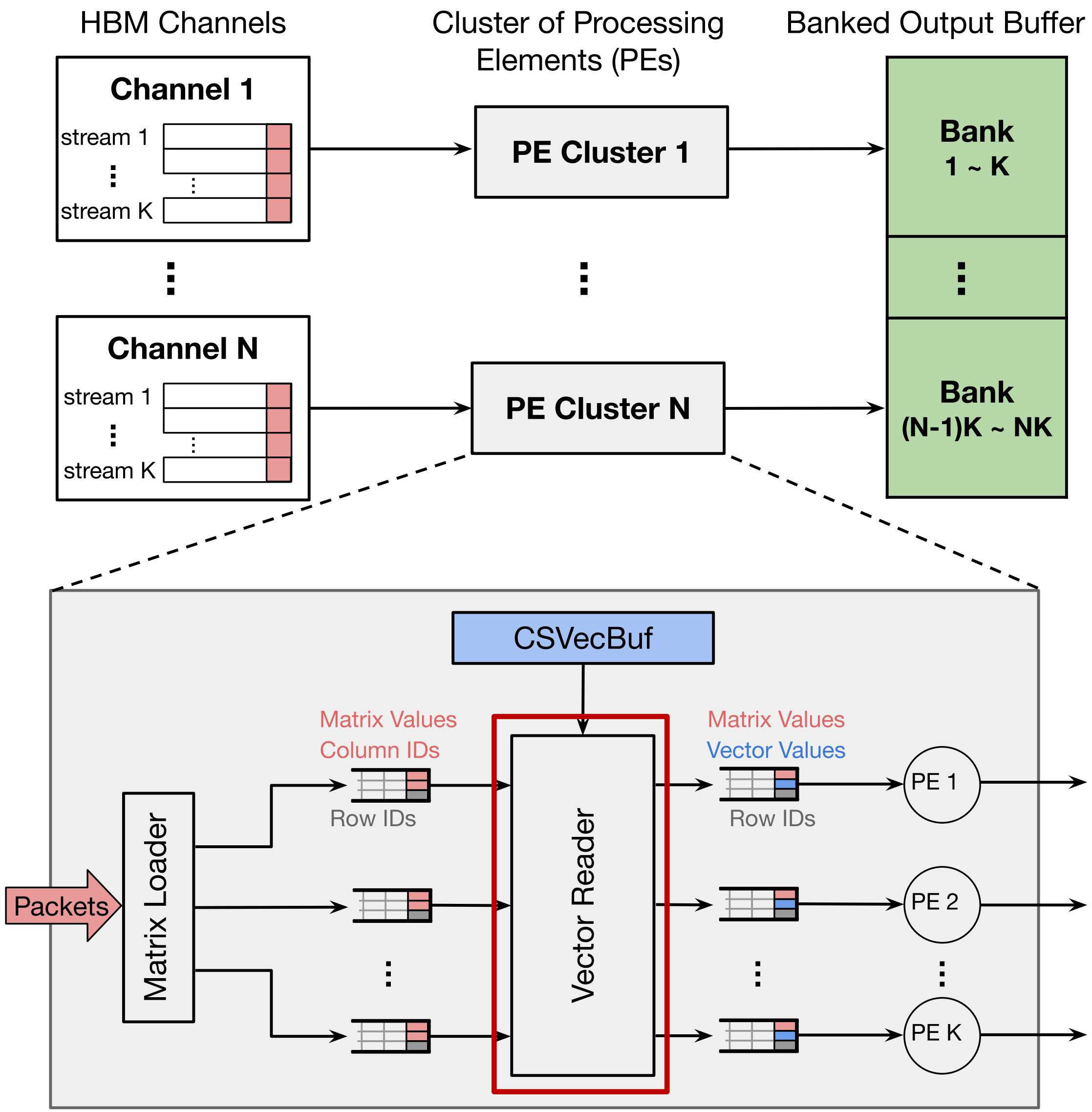
SpMV Accelerator Architecture



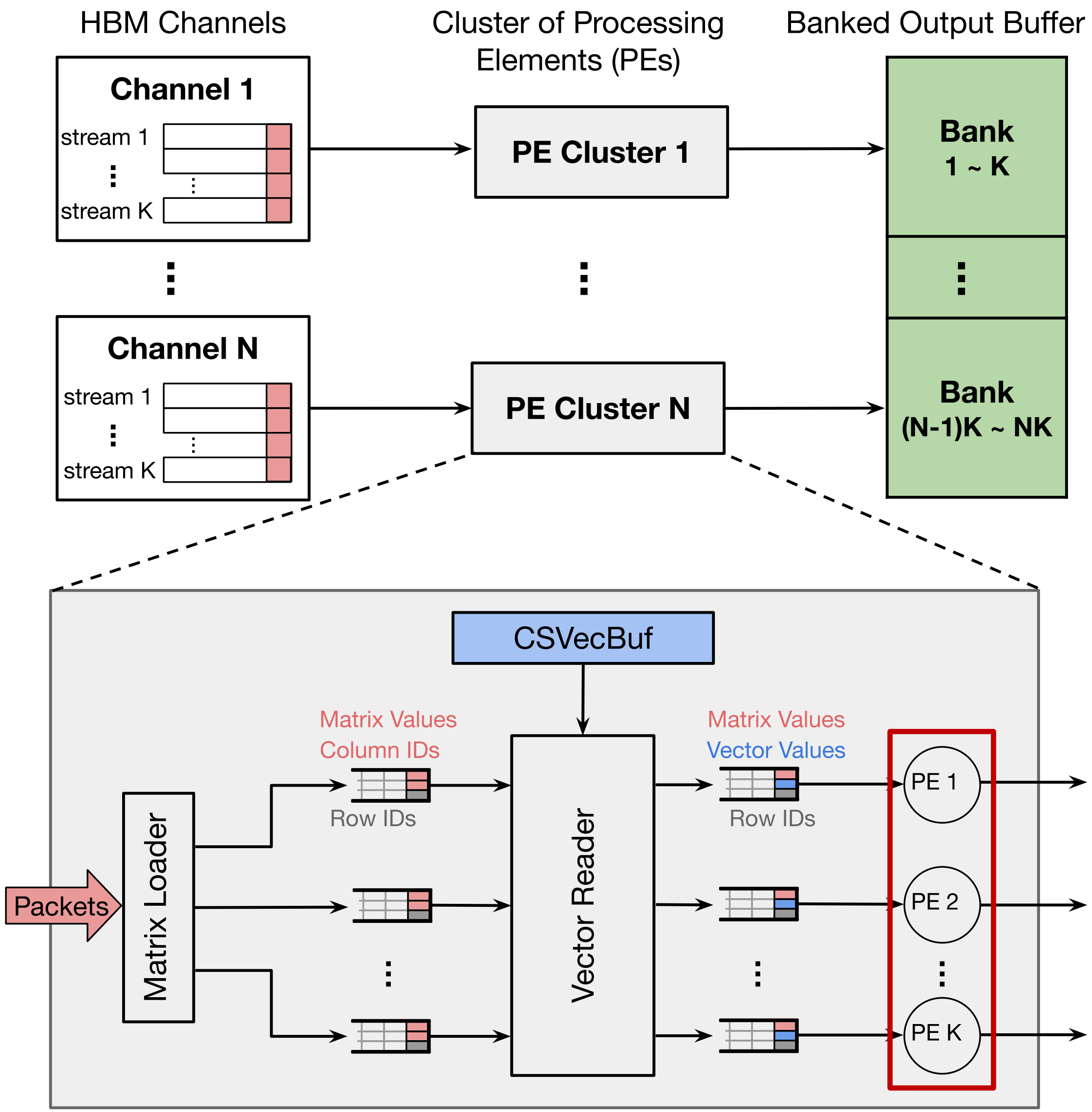
SpMV Accelerator Architecture



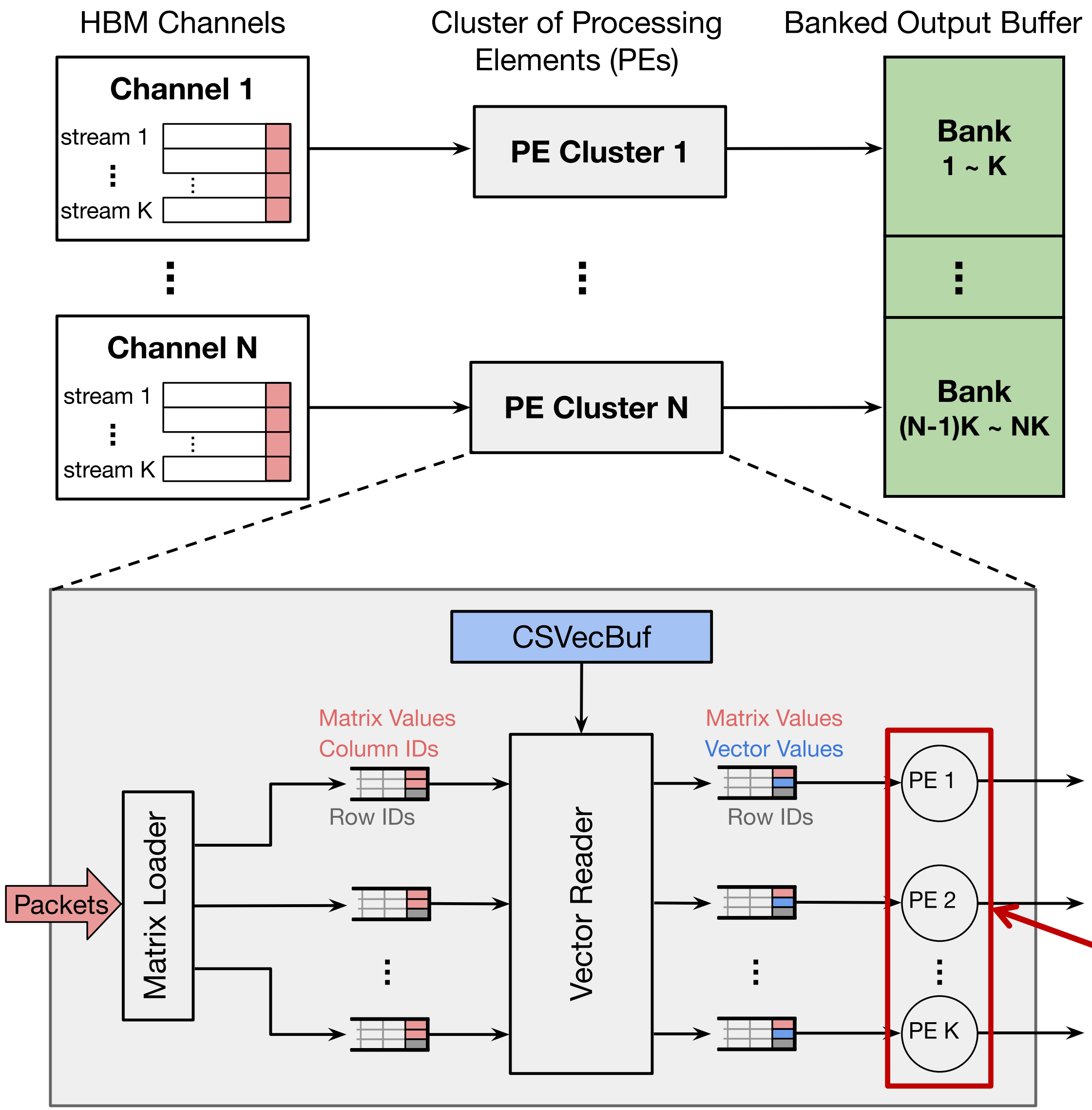
SpMV Accelerator Architecture



SpMV Accelerator Architecture

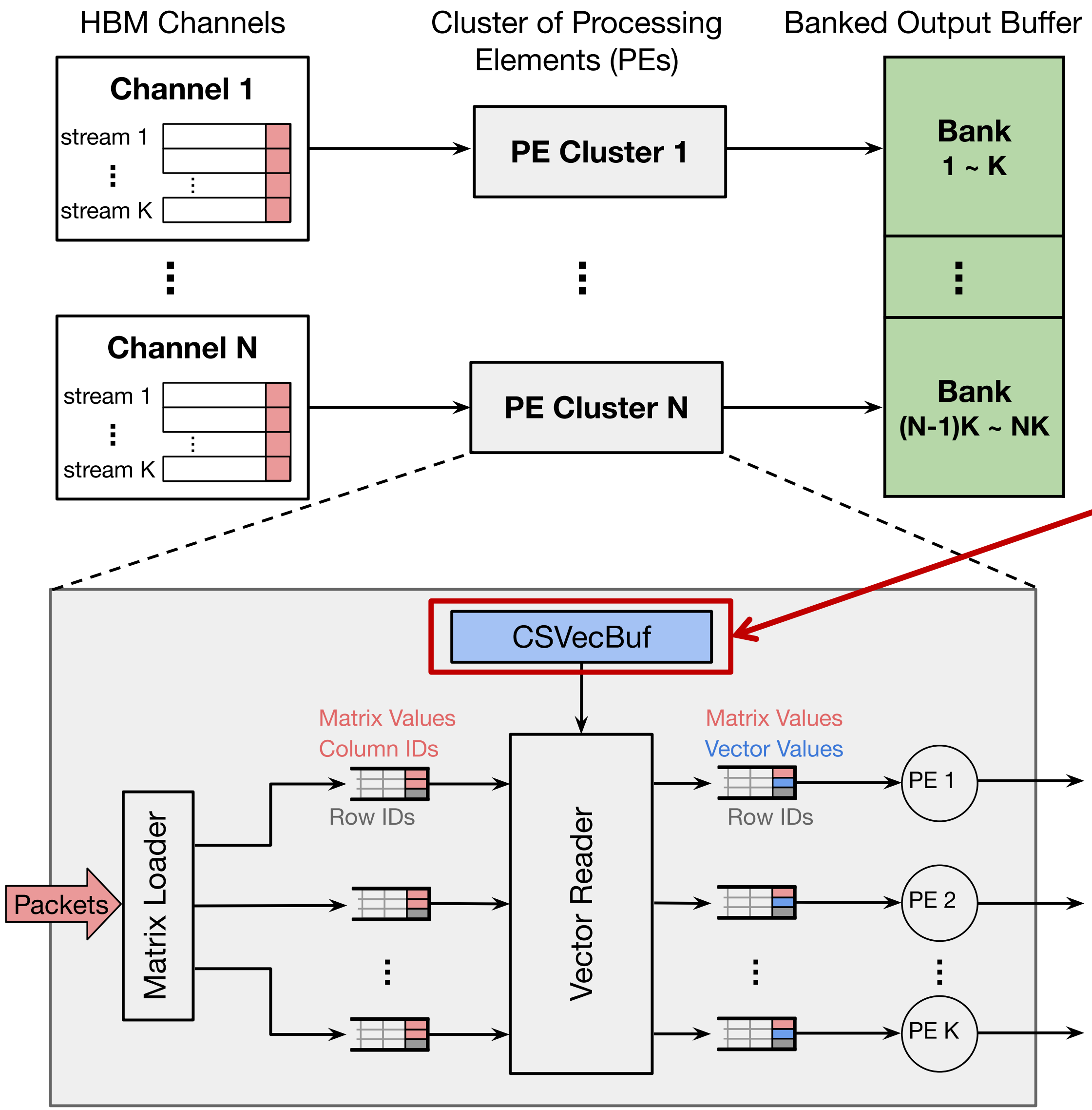


SpMV Accelerator Architecture



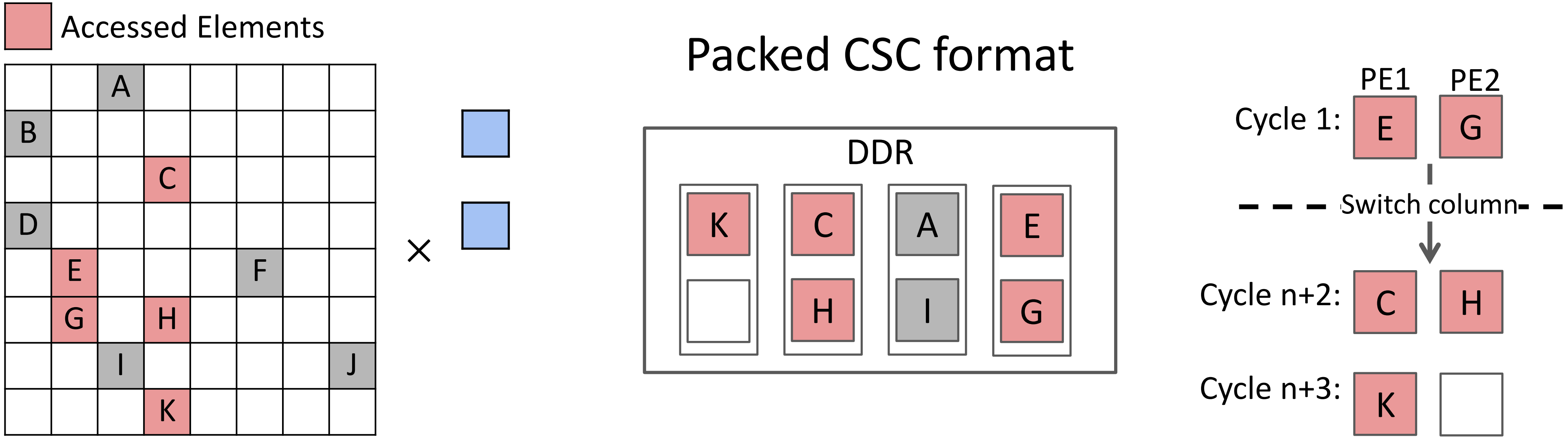
PEs support multiple semirings to handle multiple graph algorithms

SpMV Accelerator Architecture



CSVecBuf (Cluster Shared Vector Buffer) is scalable by combining vector replication and banking

SpMSpV



- Memory accesses in the same column are sequential
- Switching columns incurs random accesses
- PEs process one column at a time
  - Avoiding contention on updating the output
  - Limiting the degree of parallelization

Middleware

- Each accelerator is exposed to users as a module
- Users construct graph algorithms by specifying and scheduling the modules

```
DenseVec bfs(SparseMatrix Adj, int src, int num_iter) {
  // Initialize the frontier vector
  SparseVec frontier = {src};
  // Initialize the distance vector
  DenseVec distance(Adj.num_rows);
  for (int i=0; i<Adj.num_rows; i++) {distance[i] = 0;}
  distance[src] = 0;
  for (int iter=1; iter<=num_iter; iter++) {
    // Perform graph traversal using SpMV
    frontier = graphblast::SpMV<BoolSemiring>(Adj,
                                              frontier,
                                              distance);

    // Update distance
    graphblast::Assign(distance, frontier, iter);
  }
  return distance;
}
```

Pull-mode BFS in GraphBLAST

```
class BFS : graphlily::ModuleCollection {
  // Specify the modules and load the bitstream
  void init() {
    this->SpMV = graphlily::SpMVModule<BoolSemiring>;
    this->Assign = graphlily::AssignModule;
    load_bitstream("graphlily_overlay.bitstream");
  }
  // Format the matrix and send it to the device
  void prepare_matrix(SparseMatrix Adj) {
    AdjCPSR = this->SpMV.format(Adj);
    this->SpMV.to_hbm(AdjCPSR);
  }
  // Compute BFS by scheduling the modules
  // The logic is the same as in GraphBLAST
  DenseVec run(int src, int num_iter) {
    . . .
  }
};
```

Pull-mode BFS in GraphLily

Frequency and Resource Utilization

Implementation on a Xilinx Alveo U280 FPGA using Vitis HLS:

- 16 HBM channels for the CPSR sparse matrix
- 3 HBM channels for the input vector, the mask vector, and the output vector
- 1 DDR4 channel for the packed CSC matrix
- Total bandwidth is **285 GB/s**
- Frequency is **165 MHz**

	LUT	FF	DSP	BRAM	URAM
BFS-only	335K (30.0%)	426K (18.4%)	179 (2.0%)	393 (22.9%)	512 (53.3%)
overlay	399K (35.8%)	467K (20.2%)	723 (8.0%)	393 (22.9%)	512 (53.3%)

- Compared with BFS-only (i.e., the PEs only support the Boolean semiring), overlay consumes slightly more LUT, FF, and DSP resources

SpMV — Throughput and Bandwidth Efficiency

CPU evaluation: 32-core Intel Xeon Gold 6242 with **282 GB/s** bandwidth
GPU evaluation: GTX 1080 Ti with **484 GB/s** bandwidth

	Throughput (MTEPS)			Bandwidth efficiency (MTEPS/(GB/s))		
Dataset	MKL (32 threads)	cuSPARSE	GraphLily	MKL (32 threads)	cuSPARSE	GraphLily
googleplus	2542	13643	7002	9.0	28.2	24.6
ogbl-ppa	2065	9007	8492	7.3	18.6	29.8
hollywood	2202	11277	8736	7.8	23.3	30.7
pokec	1504	5271	4064	5.3	10.9	14.3
ogbn-products	1556	2501	6434	5.5	5.2	22.6
orkut	1807	5332	6973	6.4	11.0	24.5
Geometric mean	**1912**	**6783**	**6751**	**6.8**	**14.0**	**23.7**

- Throughput (geo-mean): 3.5× higher than MKL; comparable to cuSPARSE
- Bandwidth efficiency (geo-mean): 3.5× higher than MKL; 1.7× higher than cuSPARSE

SpMSpV — Execution Time

Execution time (ms) with different vector sparsities
Results better than SpMV are marked in green

Dataset	SpMV	SpMSpV			
		99%	99.5%	99.9%	99.95%
googleplus	2.0	4.2	3.0	1.3	0.8
ogbl-ppa	5.5	34.8	20.6	5.5	2.9
hollywood	12.9	69.9	41.1	11.5	6.4
pokec	7.5	83.9	43.5	9.6	5.2
ogbn-products	19.2	215.0	115.6	25.7	13.5
orkut	30.5	316.9	172.2	39.7	20.2

- SpMSpV outperforms SpMV when the vector sparsity is higher than 99.9%
  - We use 99.9% as the threshold in the scheduling of graph algorithms
- Future work: enhance SpMSpV

Graph Algorithms — Comparing with CPU/GPU Systems

PageRank

Throughput (MTEPS)

Dataset	GraphIt	GraphBLAST	GraphLily
googleplus	3452	7635	6252
ogbl-ppa	3622	6274	7092
hollywood	2663	8127	7471
pokec	1793	3522	2933
ogbn-products	1093	2536	5290
orkut	2151	4181	5940
Geometric mean	**2280**	**4940**	**5591**

- Throughput (pull): 2.5× higher than GraphIt; 1.1× higher than GraphBLAST

Graph Algorithms — Comparing with CPU/GPU Systems

BFS

Throughput (MTEPS)

	GraphIt		GraphBLAST		GraphBLAST	
	Pull	Pull-Push	Pull	Pull-Push	Pull	Pull-Push
googleplus						
ogbl-ppa						
hollywood						
pokec						
ogbn-products						
orkut						
Geometric mean						

Graph Algorithms — Comparing with CPU/GPU Systems

BFS

Throughput (MTEPS)

	GraphIt		GraphBLAST		GraphBLAST	
	Pull	Pull-Push	Pull	Pull-Push	Pull	Pull-Push
googleplus	2296		5804		4626	
ogbl-ppa	3047		5482		4460	
hollywood	2086		7067		5202	
pokec	1886		3140		1539	
ogbn-products	1125		2409		3419	
orkut	1816		2851		3737	
Geometric mean	**1957**		**4114**		**3581**	

- Throughput (pull): 1.8× higher than GraphIt; 10% lower than GraphBLAST

Graph Algorithms — Comparing with CPU/GPU Systems

BFS

Throughput (MTEPS)

	GraphIt		GraphBLAST		GraphBLAST	
	Pull	Pull-Push	Pull	Pull-Push	Pull	Pull-Push
googleplus	2296	3615	5804	9378	4626	4999
ogbl-ppa	3047	5279	5482	7117	4460	5111
hollywood	2086	3475	7067	10450	5202	6863
pokec	1886	2960	3140	4222	1539	1965
ogbn-products	1125	1422	2409	2799	3419	3644
orkut	1816	3201	2851	4900	3737	4937
Geometric mean	**1957**	**3103**	**4114**	**5857**	**3581**	**4286**
		1.6×		**1.4×**		**1.2×**

Power (Watt)

GraphIt	GraphBLAST	GraphLily
264	146	45

Energy efficiency (MTEPS/Watt)

GraphIt	GraphBLAST	GraphLily
11.8	40.1	95.2

- Throughput (pull): 1.8× higher than GraphIt; 10% lower than GraphBLAST
- Switching from pull to pull-push, GraphLily achieves 1.2× speedup, less significant than GraphIt (1.6×) and GraphBLAST (1.4×)
- Energy efficiency (pull-push): 8.1× higher than GraphIt; 2.4× higher than GraphBLAST

Graph Algorithms — Comparing with Single-Purpose FPGA Accelerators

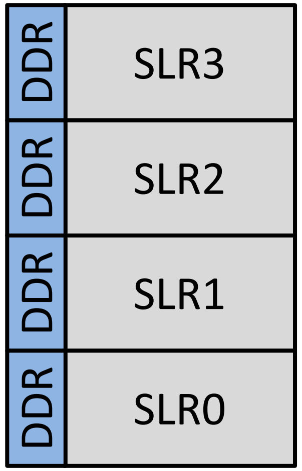
ThunderGP: measured on Alveo U250
HitGraph: simulated results
Both target DDR-equipped FPGAs

Algorithm	Dataset	System	Throughput (MTEPS)	Speedup
BFS	hollywood	ThunderGP	5960	1.2×
		GraphLily	6863	
PageRank	hollywood	ThunderGP	4073	1.8×
		GraphLily	7471	
	rmat21	HitGraph	3410	1.4×
		GraphLily	4653	
SSSP	hollywood	ThunderGP	4909	1.9×
		GraphLily	9340	
	rmat21	HitGraph	4304	1.3×
		GraphLily	5646	

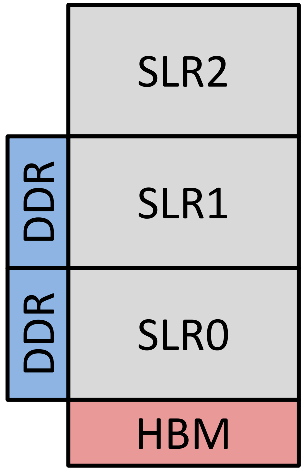
- Throughput: 1.3× to 1.4× higher than HitGraph; 1.2× to 1.9× higher than ThunderGP
- Frequency: lower than ThunderGP (165 MHz vs. 250 MHz for BFS, 243 MHz for PageRank, 251 MHz for SSSP)

Why Is the Frequency of GraphLily Lower than ThunderGP?

The heterogeneous architecture of U280 causes severe congestion on SLR0



Homogeneous architecture of U250



Heterogeneous architecture of U280

Research question: how to increase the frequency of large-scale HLS designs on multi-SLR HBM-equipped FPGAs?

GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs

Yuwei Hu, Yixiao Du, Ecenur Ustun, Zhiru Zhang

Cornell University

https://github.com/cornell-zhang/GraphLily

Cornell University