Near-Memory Hardware Specialization for Fast and Efficient Sparse Processing

Zhiru Zhang
School of ECE, Cornell University

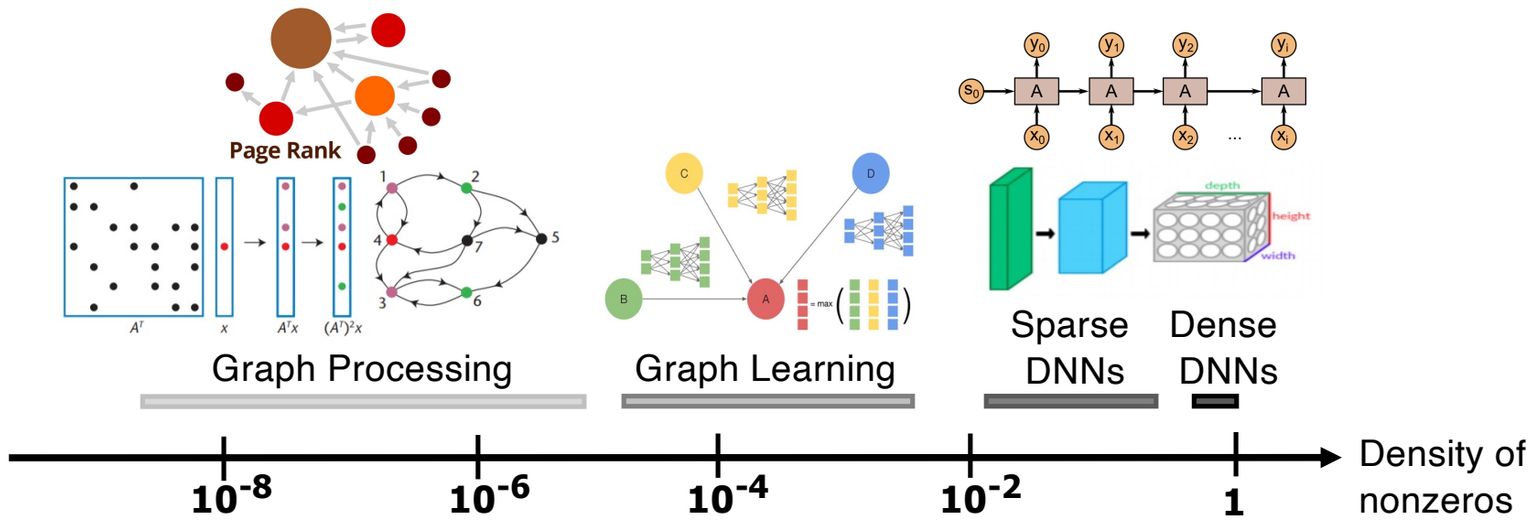
DATA-DREAM Workshop @ DATE, 3/18/2022

Cornell University

CSL

The Game Changing Potential of Sparsity

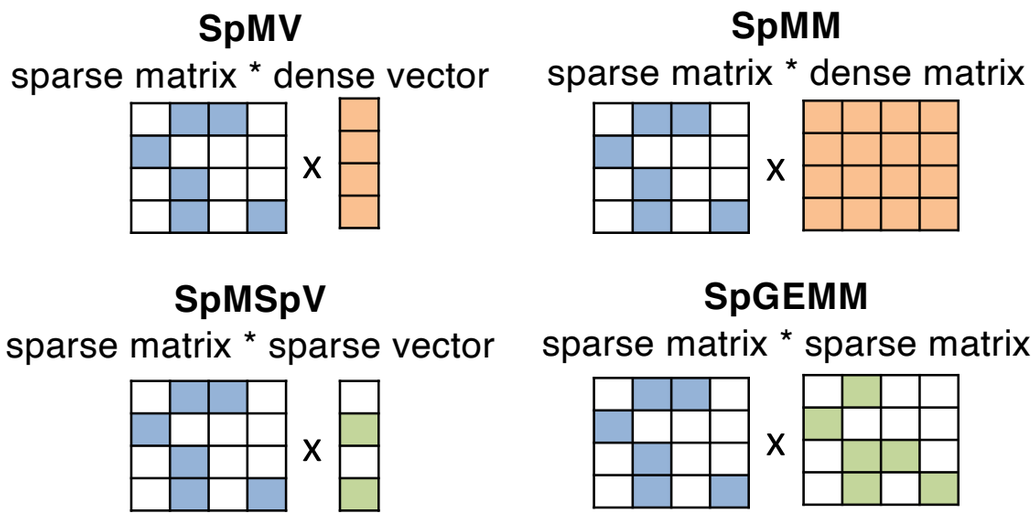
- **Opportunity:** Sparsity reduces the amount of data that needs to be processed by orders of magnitude

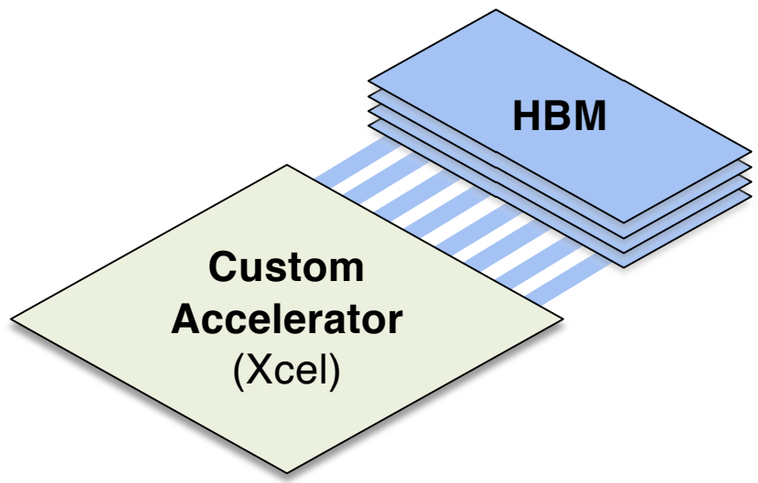


- **Challenge:** Realistic sparse workloads are highly memory bound, exhibiting varying sparsity and irregular patterns in compute & data access

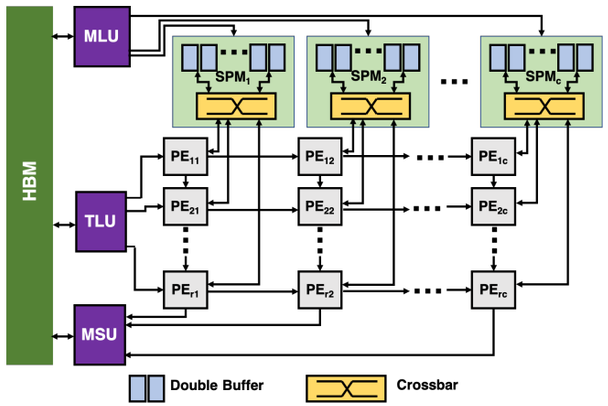
Our Approach to Efficient Sparse Processing

- Near-memory hardware acceleration for common sparse linear algebra operations (aka ‘motifs’) by
  - exploiting high-bandwidth memories (HBMs)
  - co-design of sparse format and accelerator architectures
  - prototyping on FPGAs with high-level synthesis (HLS)

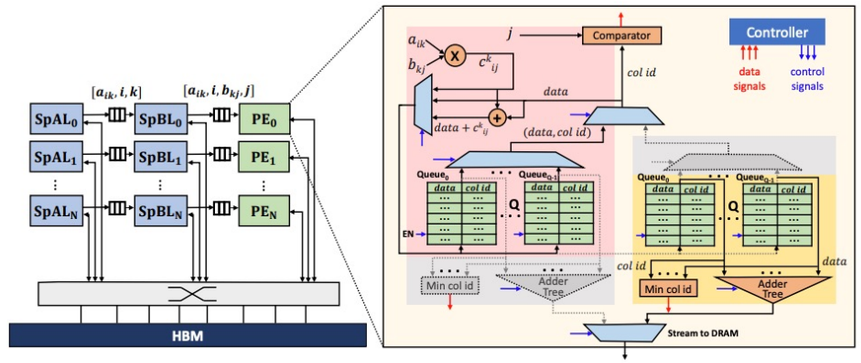




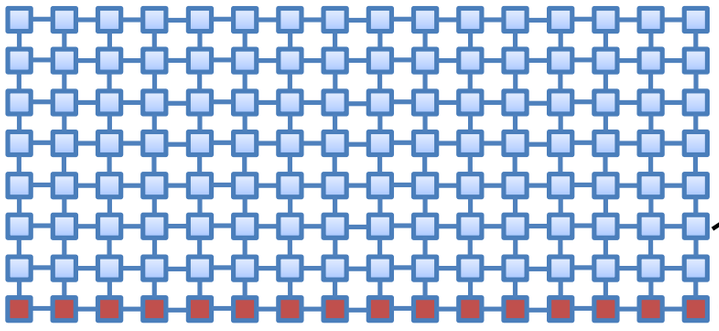
Our Recent Work on Efficient Sparse Accelerators



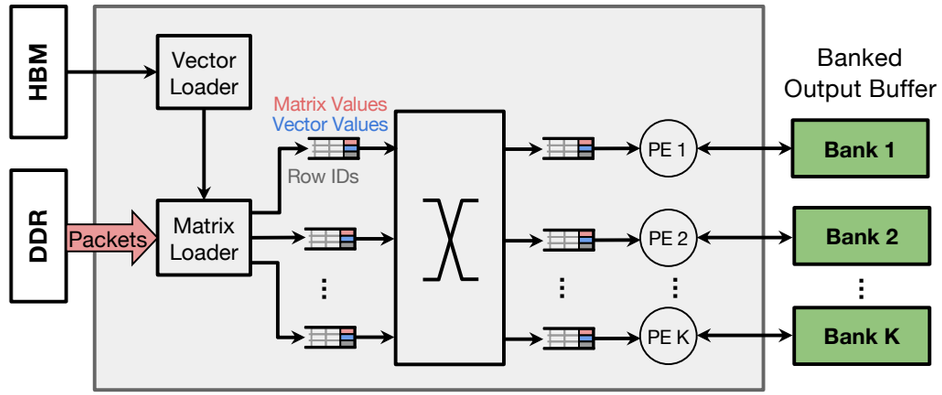
Tensaurus: Sparse-Dense Matrix/Tensor Xcel [HPCA'20]



MatRaptor: Sparse-Sparse MatMul Xcel [MICRO'20]



Manycore for Dense/Sparse Tensor Processing [TCAD'21]



FPGA-based Sparse Accelerators [ICCAD'21, FPGA'22]

Focus of this talk

Agenda

HiSparse: SpMV Acceleration on HBM-Equipped FPGAs

GraphLily: FPGA-Based Graph Linear Algebra Overlay

Agenda

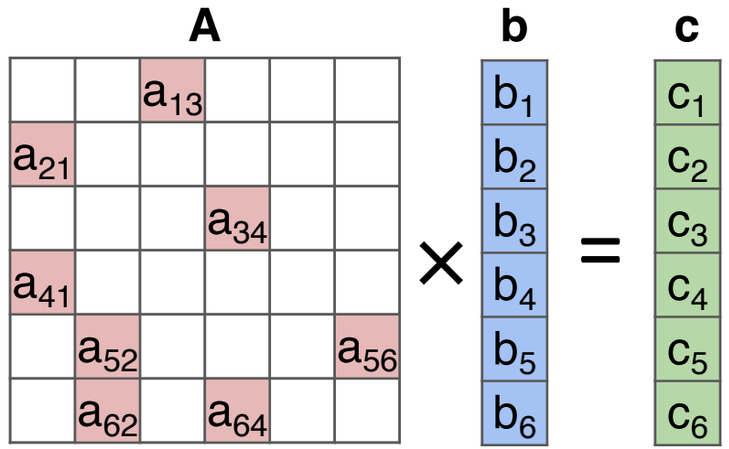
HiSparse: SpMV Acceleration on HBM-Equipped FPGAs

Int'l Symposium on Field-Programmable Gate Arrays (FPGA 2022)
in collab. with: Yixiao Du, Yuwei Hu


github.com/cornell-zhang/HiSparse

GraphLily: FPGA-Based Graph Linear Algebra Overlay

Characteristics of SpMV



Sparse Matrix-Vector Multiplication (SpMV)

$c_1 = a_{13} \times b_3 + c_1$

$c_2 = a_{21} \times b_1 + c_2$

$c_3 = a_{34} \times b_4 + c_3$

$c_4 = a_{41} \times b_1 + c_4$

$c_5 = a_{52} \times b_2 + c_5$

$c_5 = a_{56} \times b_6 + c_5$

2 arithmetic operations
4 memory accesses

elements of A used only once

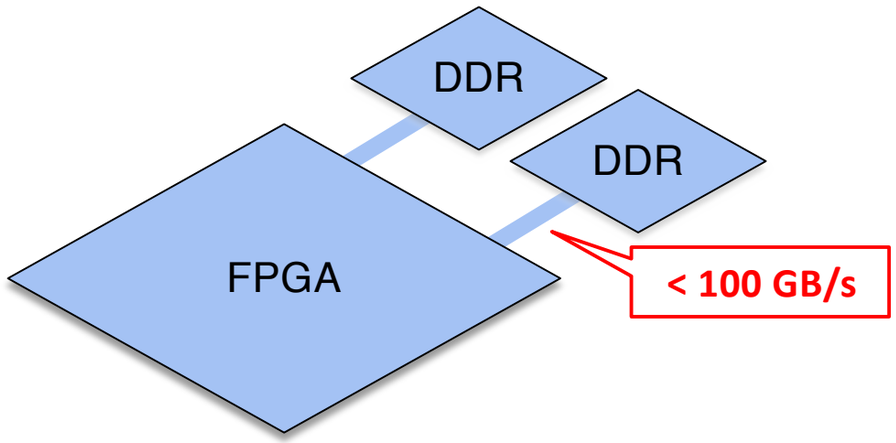
random access to b

read-after-write dependency

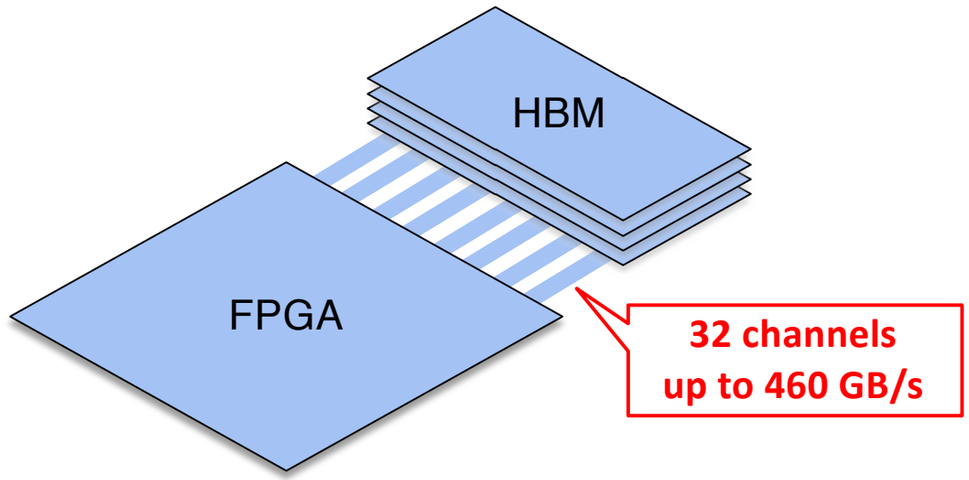
- Low compute to memory access ratio (or operational intensity)
- Irregular data access patterns
- Carried data dependencies

⇨ Memory (bandwidth) bound

Target Acceleration Platform



- Previous work targeted systems using low-bandwidth DDRs [1-4]

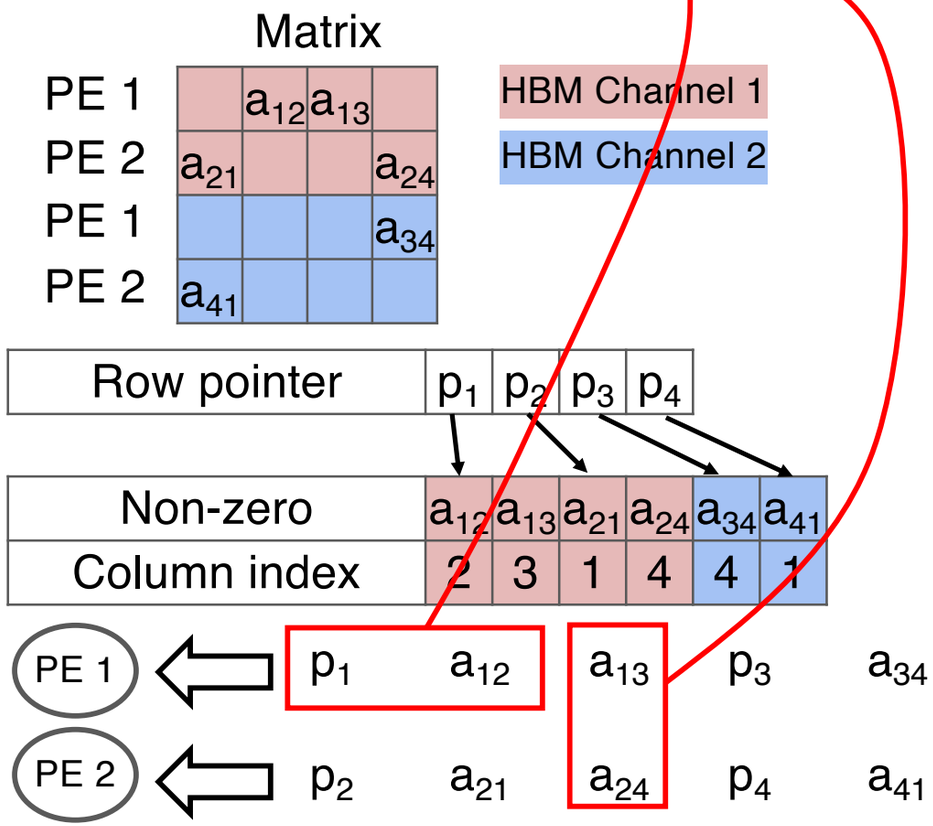


- HBM-equipped (multi-die) FPGAs
  - Much higher bandwidth available with concurrent accesses to multiple channels
  - Important to avoid channel conflicts to achieve high bandwidth utilization

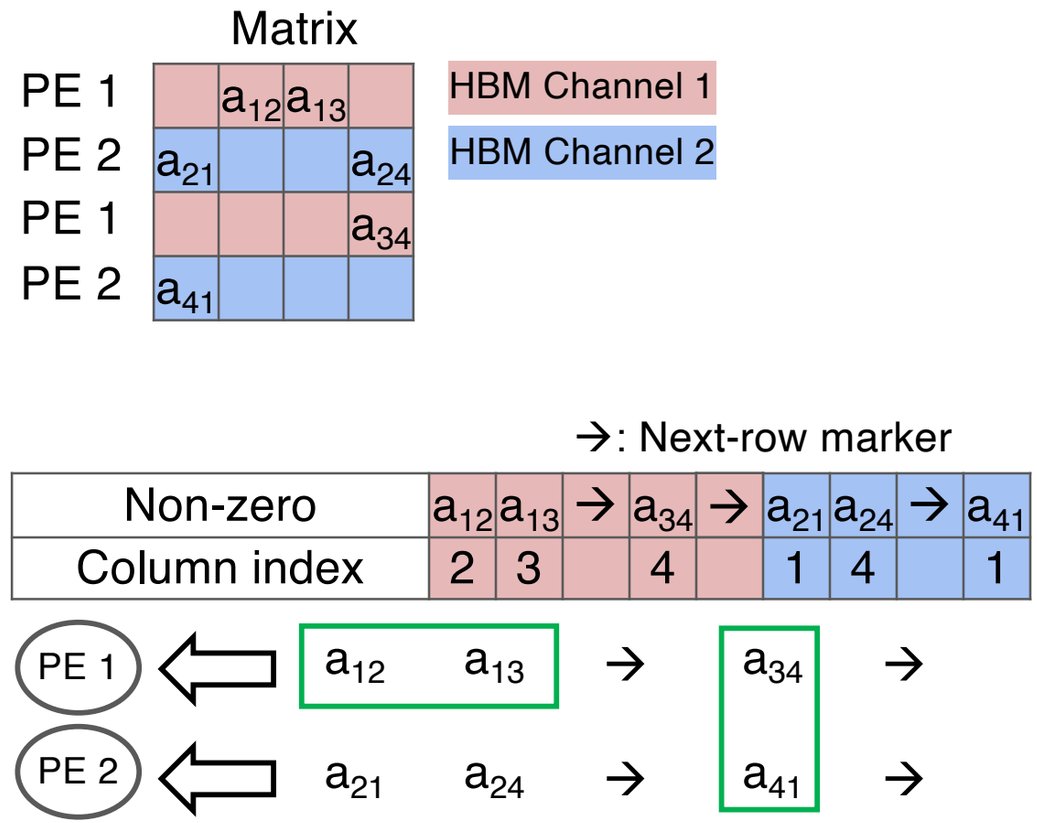
[1] S. Kestur, et al. Towards a universal FPGA matrix-vector multiplication architecture, FCCM'12
[2] J. Fowers, et al. A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication, FCCM'14
[3] T. Geng, et al. AWB-GCN: A graph convolutional network accelerator with runtime workload balancing. MICRO'20
[4] X. Chen, et al. ThunderGP: HLS-based graph processing framework on FPGAs, FPGA'21

Sparse Matrix Formats

- Compressed Sparse Row (CSR)
  - Non-streaming access
  - Channel conflicts

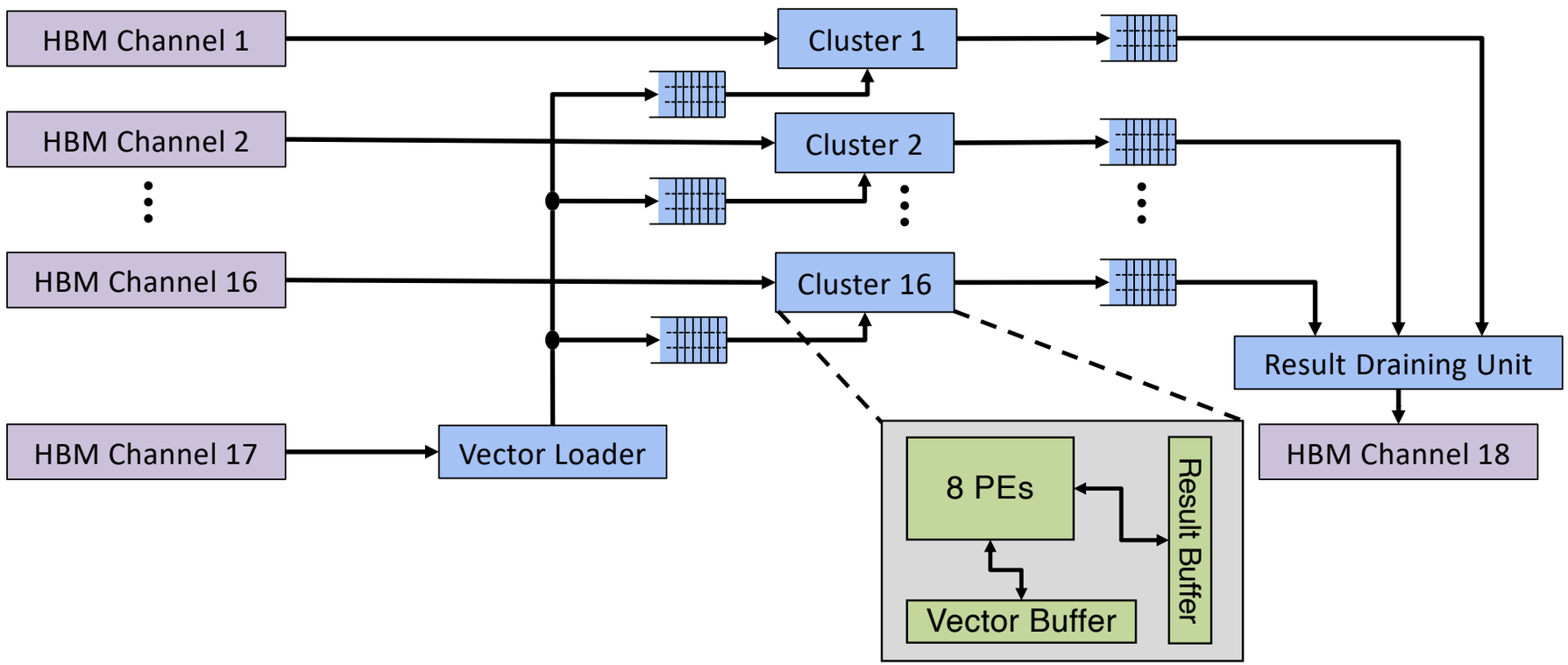


- Accelerator-Friendly Format (CPSR)
  - Streaming (vectorized) access
  - No channel conflicts



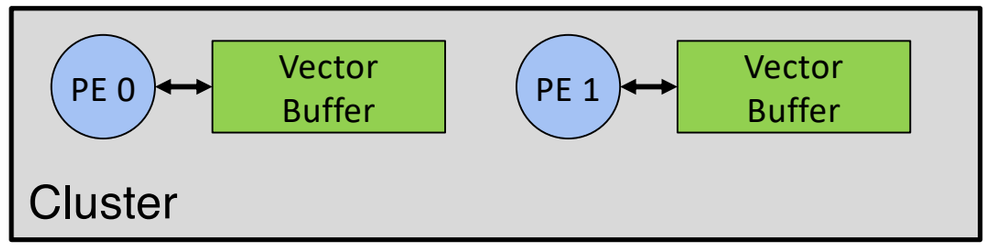
HiSparse Accelerator Architecture

- Parallel processing engines (PEs) grouped in clusters
  - Each cluster accesses a dedicated HBM channel
    - Multiple PEs per a cluster for vectorized HBM access
  - Each cluster requires on-chip buffers for storing dense vector and results

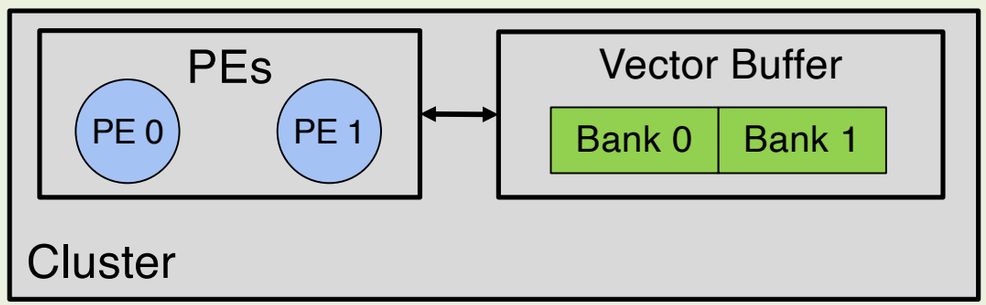


On-Chip Buffering of the Dense Vector

- Option 1: Duplicating dense vector to each PE
  - High on-chip bandwidth
  - High demand for on-chip memory

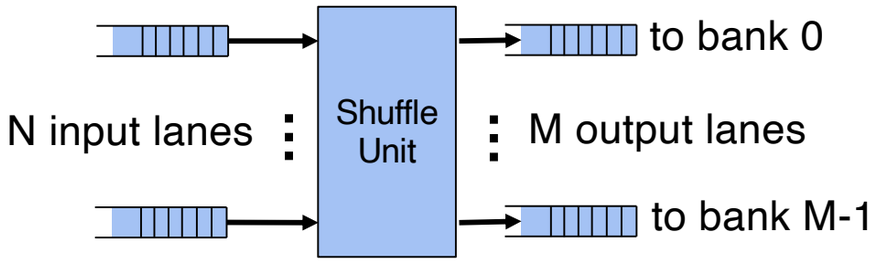


- Option 2: Sharing dense vector among PEs
  - Efficient use of on-chip memory
  - Bank conflicts



A shuffle unit is required to manage data requests and bank conflicts

Handling Bank Conflicts in HLS




```
/* shuffle unit counter-example */
// N is the number of input lanes
while (!exit) {
# pragma HLS pipeline // II will be N
    for (int i = 0; i < N; i ++) {
# pragma HLS unroll
        // fetch a payload
        payload = in_lane[i].read();
        // send the payload to its target
        out_lane[payload.target].write(payload);
    }
}
```

HLS assumes the worst-case traffic pattern: *N* input lanes

→ Initiation Interval (II) = *N*

```
/* shuffle unit */
/* N is the number of input lanes
   M is the number of output lanes
   ARB_DEPTH is the pipeline depth for the arbiter */
while (!exit) {
# pragma HLS pipeline // II will be 1
# pragma HLS dependence variable=granted inter RAW \
true distance=ARB_DEPTH
# pragma HLS dependence variable=arbiter_out inter \
RAW true distance=ARB_DEPTH
    for (int i = 0; i < N; i ++) {
# pragma HLS unroll
      if (granted[i]) arbiter_in[i] = in_lane[i].read();
      else arbiter_in[i] = arbiter_out[i];
    }
    arbiter(arbiter_in, sel, granted, arbiter_out);
    for (int i = 0; i < M; i ++) {
# pragma HLS unroll
      if (granted[sel[i]])
          out_lane[i].write(arbiter_out[sel[i]]);
    }
}
```

Explicit arbitration & re-sending: II = 1

Handling Carried Dependencies in HLS

Naïve Implementation

```
/* carried dependency counter-example */
// X is the read latency; Y is the write latency
while (!exit) {
# pragma HLS pipeline // II will be X + Y
    // fetch a payload
    payload = in_lane.read();
    // multiply accumulation
    result_buffer[payload.row_index]
        += payload.mat_value * payload.vec_value;
}
```

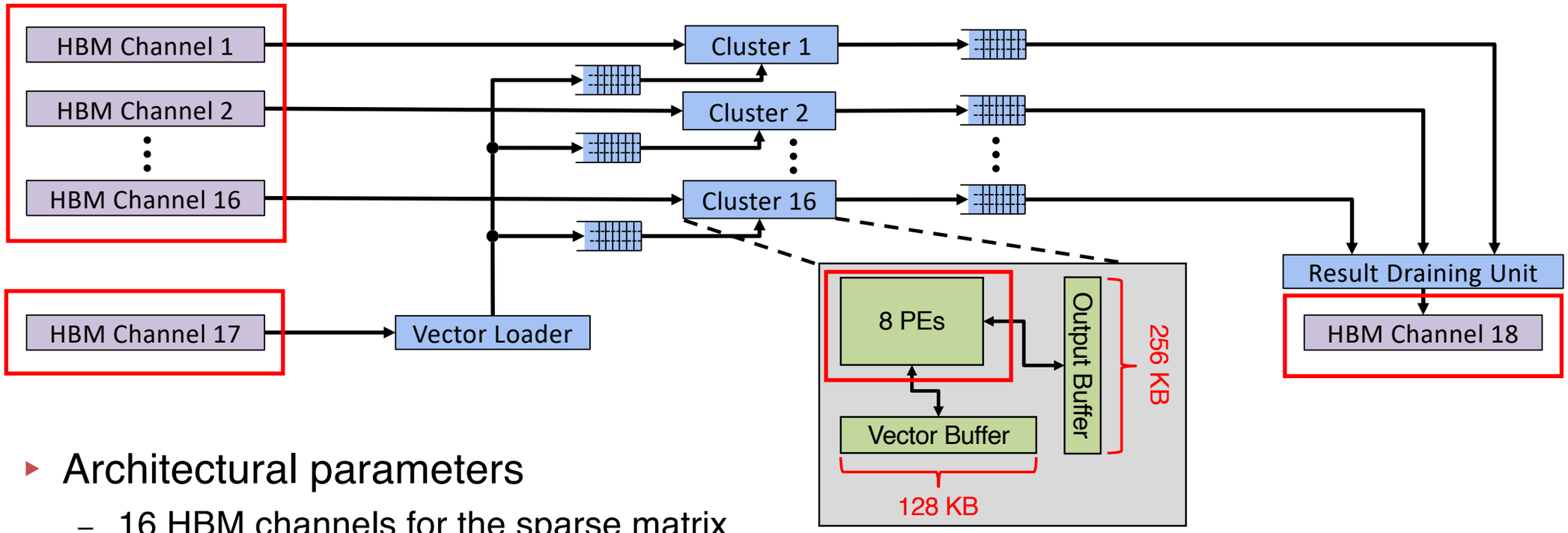
- Low throughput with a large II
- Challenging to implement data forwarding in untimed HLS
  - Pipeline registers are not visible at the source level

Optimized Implementation

```
/* data forwarding */
// X is the read latency; Y is the write latency
/* in-flight write queue (IFWQ) is
   a shift register of depth (X + Y) */
while (!exit) {
# pragma HLS pipeline // II will be 1
# pragma HLS dependence variable=result_buffer false
    // fetch a payload
    payload = in_lane.read();
    // read buffer
    old_val = result_buffer[payload.row_index];
    // data forwarding
    val = detect_raw(IFWQ, payload.row_index) ?
          IFWQ.find_val(payload.row_index)] : old_val;
    // multiply and add
    new_val = val
            + payload.mat_value * payload.vec_value;
    // write buffer
    result_buffer[payload.row_index] = new_val;
    // update IFWQ (shift register)
    IFWQ.update(new_val, payload.row_index)
}
```

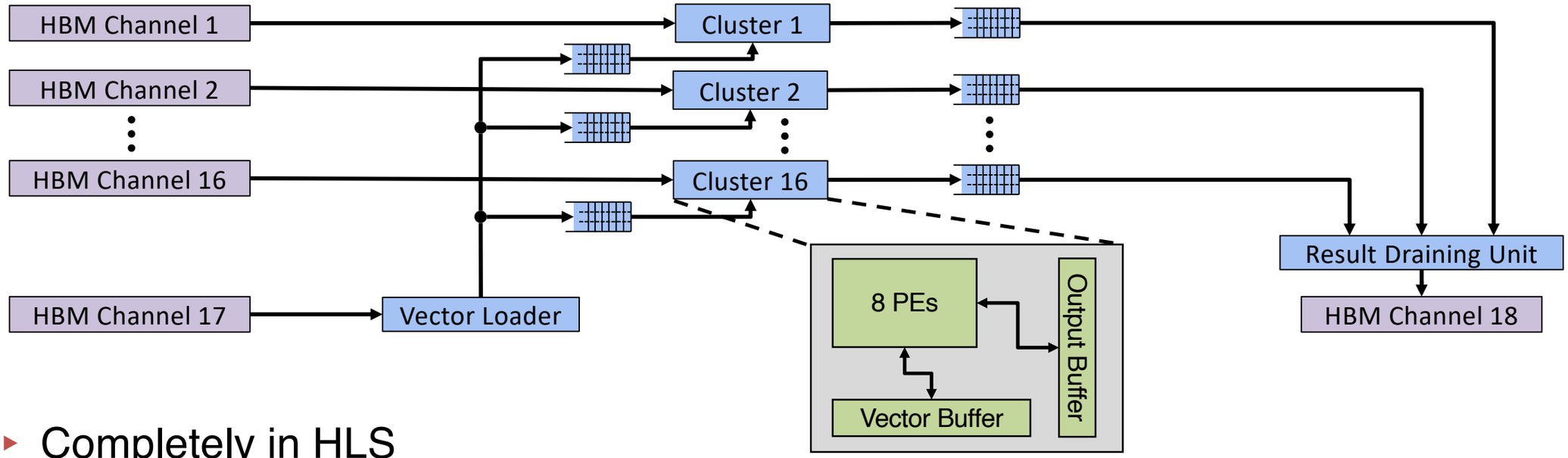
- IFWQ stores writes from past iterations
  - Behaves like pipeline registers

Implementation on Xilinx (AMD) Alveo U280 FPGA



- Architectural parameters
  - 16 HBM channels for the sparse matrix
  - 2 HBM channels for input/result vector
  - Total bandwidth (of 18 channels): 258 GB/s
  - 4 MB result buffer (256 KB x 16), 128 KB vector buffer (16 copies)
  - 8 process engines per cluster, 128 PEs in total

Implementation on Xilinx (AMD) Alveo U280



- Completely in HLS
- Frequency: 237 MHz with floorplanning
- Resource Utilization:

LUT	FF	DSP	BRAM	URAM
544K (47.3%)	528K (22.7%)	688 (7.63%)	128 (7.22%)	512 (53.3%)

HiSparse vs. CPU & GPU Implementations (1)

- CPU baseline
  - Intel MKL (32 threads)
  - Intel Xeon Gold 6242 with 282 GB/s bandwidth
- GPU baseline
  - NVIDIA cuSPARSE
  - GTX 1080 Ti with 484 GB/s bandwidth

Throughput (GOPS)							
Dataset	MKL	cuSPARSE	HiSparse	Dataset	MKL	cuSPARSE	HiSparse
transfomer-50	5.9	26.9	21.9	mouse-gene	12.1	29.0	27.2
transfomer-60	5.6	21.5	18.9	googlepuls	5.1	27.2	21.2
transfomer-70	5.2	17.7	16.5	ogbl-ppa	4.1	18.0	24.4
transfomer-80	4.1	19.4	14.8	hollywood	4.4	22.6	24.9
transfomer-90	2.3	13.6	9.7	pokec	3.0	10.5	11.2
transfomer-95	1.2	10.7	5.7	ogbn-products	3.1	5.0	20.6
Geomean	MKL: 4.1			cuSPARSE: 16.8		HiSparse: 16.7	

4.1x faster than MKL; (nearly) the same performance as cuSPARSE

HiSparse vs. CPU & GPU Implementations (2)

- CPU baseline
  - Intel MKL (32 threads)
  - Intel Xeon Gold 6242 with 282 GB/s bandwidth
- GPU baseline
  - NVIDIA cuSPARSE
  - GTX 1080 Ti with 484 GB/s bandwidth

Bandwidth Efficiency (MOPS/GBPS)			
Geomean	MKL: 14.0	cuSPARSE: 33.2	HiSparse: 64.5

4.6x higher than MKL; 1.9x higher than cuSPARSE

	MKL	cuSPARSE	HiSparse
Power (W)	276	153	45
Energy Efficiency (GOPS/W)	0.01	0.10	0.37

37x more efficient than MKL; 3.7x more efficient than cuSPARSE

HiSparse vs. Existing FPGA Accelerators

- ThunderGP [1]
  - Xilinx (AMD) Alveo U250 with 77 GB/s bandwidth (4 DDR), 250 MHz
- Vitis Sparse Library
  - Xilinx (AMD) Alveo U280 with 268 GB/s bandwidth (16 HBM + 2 DDR), 220 MHz

	ThunderGP#	HiSparse#	Vitis Sparse Lib*	HiSparse*
Throughput (GOPS)	8.9	**19.6**	9.4	**12.6**
Bandwidth efficiency (MOPS/GBPS)	116.0	**76.3**	34.9	**48.9**

On mouse_gene, hollywood, and pokec
* On Transformer datasets (current Vitis lib cannot handle large matrices with size beyond 100Kx100K)

2.2x faster than ThunderGP; 1.4x faster than Vitis Sparse Library

[1] Xinyu Chen, et al. ThunderGP: HLS-based graph processing framework on FPGAs, FPGA 2021.

More Recent Results

- HiSparse with 22 HBM channels
  - 20 channels for matrix, 2 channels for input/output vector
  - 231 MHz, 315 GB/s (1.22x) bandwidth

	LUT	**FF**	**DSP**	**BRAM**	**URAM**
18 channel	544K (47.3%)	528K (22.7%)	688 (7.63%)	128 (7.22%)	512 (53.3%)
22 channel	677K (59.4%)	653K (28.6%)	860 (9.53%)	157 (8.86%)	640 (66.7%)

Dataset	18 channels (ms)	**22 channels** (ms)	Speedup
pokec	5.41	4.73	**1.14x**
ogbn-products	12.87	9.32	**1.38x**
orkut	21.09	15.61	**1.35x**
LiveJournal	22.30	17.97	**1.24x**
road_usa	343.36	282.80	**1.21x**

Agenda

HiSparse: SpMV Acceleration on HBM-Equipped FPGAs

Int'l Symposium on Field-Programmable Gate Arrays (FPGA 2022)
in collab. with: Yixiao Du, Yuwei Hu

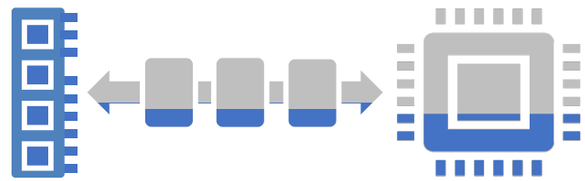
GraphLily: FPGA-Based Graph Linear Algebra Overlay

Int'l Conference On Computer Aided Design (ICCAD 2021)
in collab. with: Yuwei Hu, Yixiao Du, Ecenur Ustun

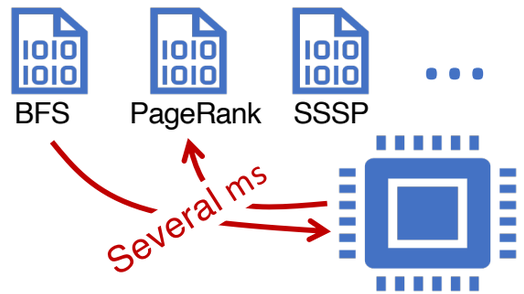

github.com/cornell-zhang/GraphLily

Graph Processing on FPGAs

- Prior efforts on FPGA-based graph processing
  (e.g., GraphGen [1], ForeGraph [2], HitGraph [3], ThunderGP [4])
  - Exploit fine-grained parallelism and custom memory hierarchy
  - Most target systems using low-bandwidth DDRs



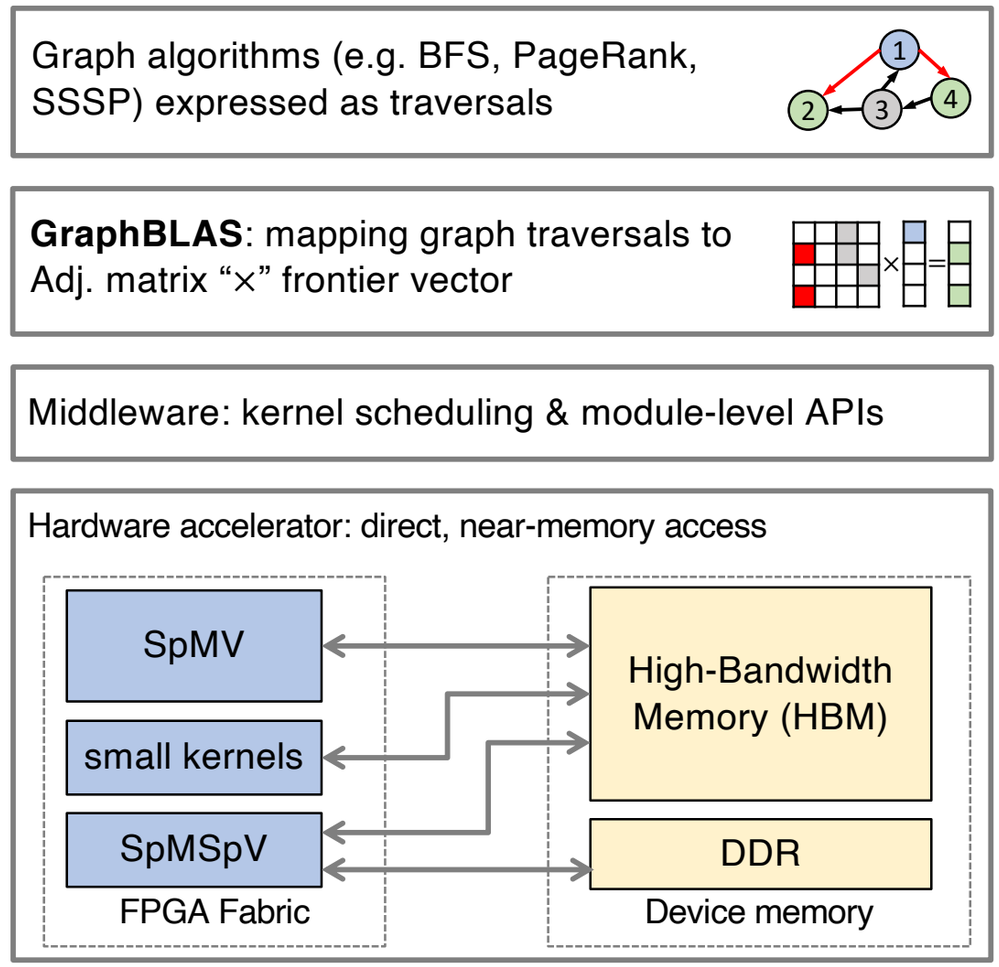
- **Key limitation** – a separate bitstream is required for each graph algorithm
  - Generating a new bitstream takes hours-days
  - The reconfiguration cost at run time is nontrivial



[1] E. Nurvitadhi, et al, An FPGA Framework for Vertex-Centric Graph Computation, FCCM'14
[2] G. Dai, et al. ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture, FPGA'17
[3] S. Zhou, et al. HitGraph: High-throughput Graph Processing Framework on FPGA, TPDS'19
[4] X. Chen, et al. ThunderGP: HLS-based Graph Processing Framework on FPGAs, FPGA'21

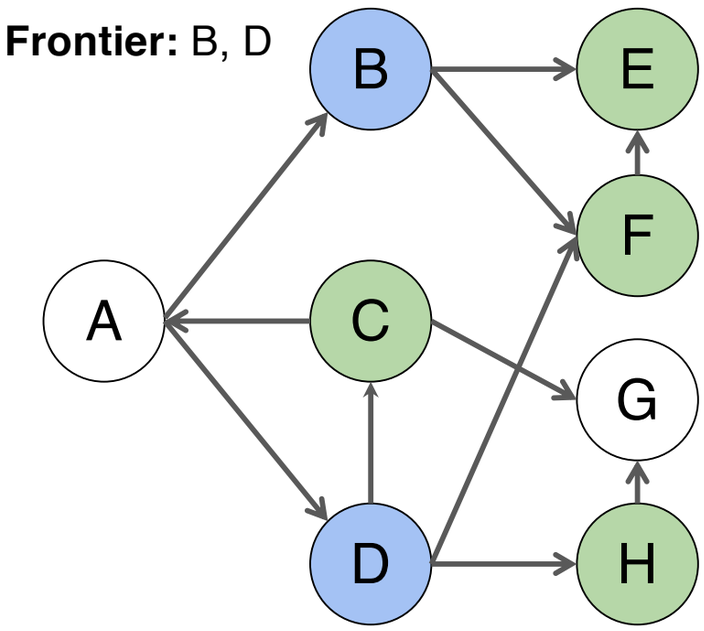
Domain-Specific Overlay for Graph Processing

- **GraphLily – the first FPGA overlay for graph linear algebra**
  - A unified & versatile accelerator that supports multiple graph algorithms
  - Effectively utilizing HBM bandwidth by co-optimizing data layout and accelerator architecture
  - Easily porting graph algorithms from CPUs/GPUs to FPGAs with a middleware



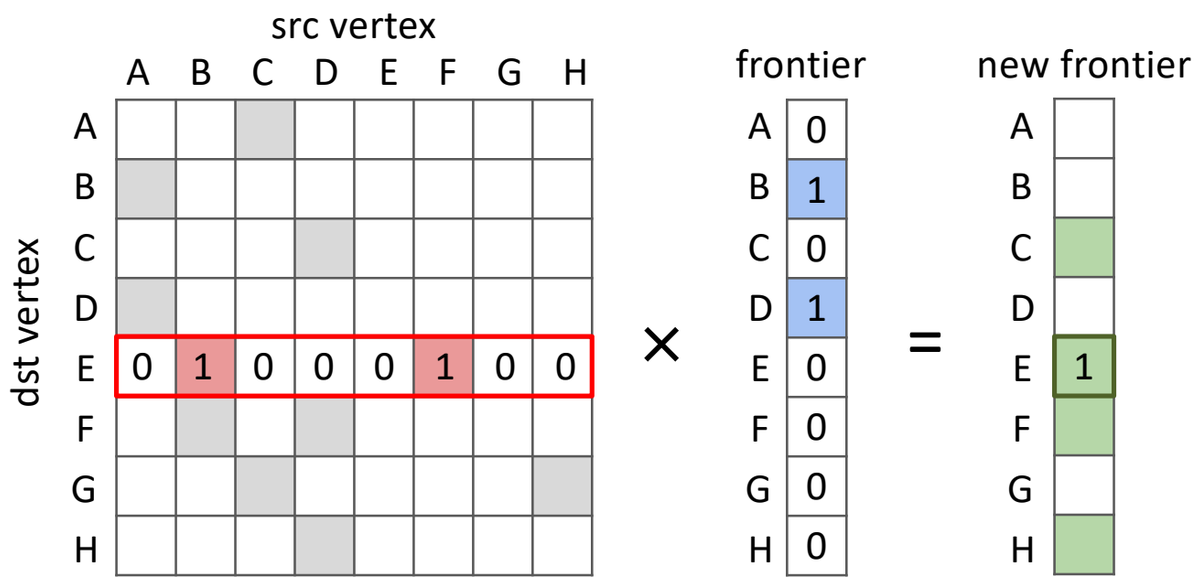
Sparse Linear Algebra Formulation of Graph Algorithms

Breadth-first search (BFS)



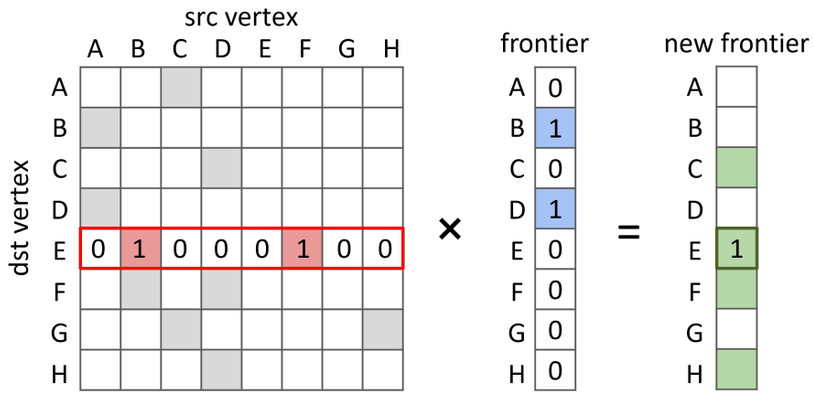
New Frontier: C, E, F, H

BFS in Matrix View



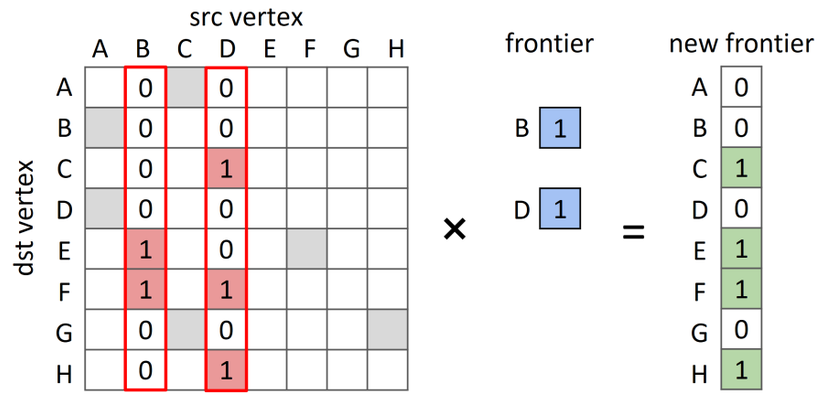
SpMV
(Adjacency matrix * frontier vector)

SpMV vs. SpMSpV



SpMV (inner product)

- More work
- Regular memory accesses
- Easy to parallelize



SpMSpV (outer product)

- Less work
- Less regular memory accesses
- More challenging to parallelize due to synchronization on output updates

Use SpMSpV when the frontier set is small
(aka direction-optimizing heuristic)

The GraphBLAS Abstraction

- GraphBLAS expresses **graph algorithms in terms of sparse linear algebra primitives**
  - Generalized matrix/vector operations over semirings

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

  - Existing implementations target 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

Programming Interface

- GraphLily provides middleware API that helps users port GraphBLAST code FPGAs

```
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;
}
```

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
  // Same logic as in GraphBLAST
  DenseVec run(int src, int num_iter) {
    . . .
  }
};
```

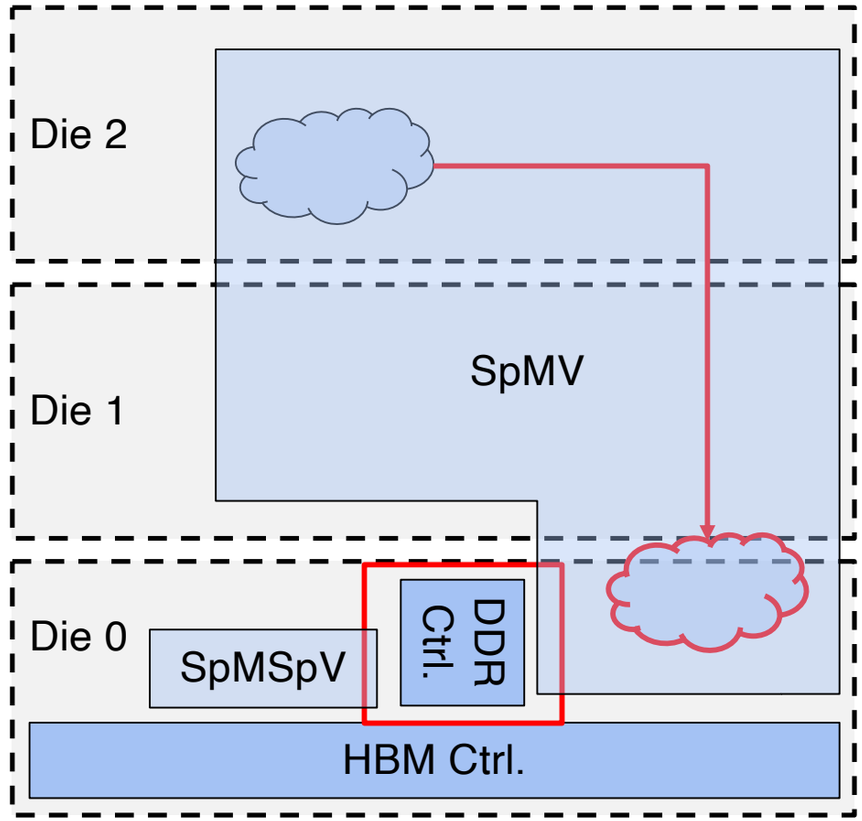
BFS in GraphLily

Resource Utilization, Layout, and Frequency

- Implementation on a Xilinx Alveo U280 FPGA using Vitis HLS
  - 19 HBM channels + DDR: Total bandwidth is 285 GB/s
  - Frequency is 165 MHz (due to congestion)
    - Integration of HiSparse is underway

	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%)

GraphLily overlay is resource efficient



Layout challenges: (1) slow cross-die communications (2) congestion around memory controllers in die 0

GraphLily vs. CPU & GPU Implementations

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: 2.5× higher than GraphIt on CPU; 1.1× higher than GraphBLAST on GPU

GraphLily vs. Single-Purpose FPGA Accelerators

Throughput in million traversed edges per second (MTEPS)

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	

- 1.3× to 1.4× higher throughput than (simulated results of) HitGraph;
  1.2× to 1.9× higher than ThunderGP

Conclusions

- Near-memory hardware specialization is a promising way for efficient sparse processing
  - Customized memory hierarchy & data layout → higher bandwidth utilization
- HBM-enabled FPGAs offer a flexible platform for implementing application- or domain-specific sparse accelerators
  - HiSparse & GraphLily can serve as useful references
- A plethora of challenges / opportunities remain:
  - HLS for bandwidth-demanding accelerators
  - Balance between efficiency and programmability
  - End-to-end acceleration of mixed sparse-dense workloads

  …