GraFlex: Flexible Graph Processing on FPGAs through Customized Scalable Interconnection Network

Chunyou Su[1], Linfeng Du[1], Tingyuan Liang[1], Zhe Lin[2], Maolin Wang[3], Sharad Sinha[4], Wei Zhang[1]

[1] The Hong Kong University of Science and Technology (HKUST)
[2] Sun Yat-sen University
[3] AI Chip Center for Emerging Smart Systems (ACCESS)
[4] Indian Institute of Technology Goa

Mar. 5, FPGA 2024

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

ACCESS
AI Chip Center for Emerging Smart Systems
智能晶片與系統研發中心

Outline

- Graph Processing on FPGAs
- Background
- GraFlex Overview
- GraFlex Design Methodology
- Evaluation

Graph Processing on FPGAs

- Designing Efficient Graph Accelerators is Challenging
  - Extremely low memory bandwidth utilization **(Memory)**
    - Irregular memory access pattern
    - Variation in memory traces among parallel PEs
  - Efficiently deploying parallel graph algorithms on hardware is difficult **(Computing)**
    - Global synchronization
    - Run-time load imbalance
- FPGA is ideal for Application-specific Architectures
  - Fast design customization
  - Flexible memory manipulation
  - High power efficiency
- Early-stage studies stick to RTL design and tend to **buffer the entire graph** on the scratchpad memory
- More recently, High-Level Synthesis (**HLS**) design methodology and **HBM** devices promote the revolution of FPGA graph-based processing

Graph Processing on FPGAs

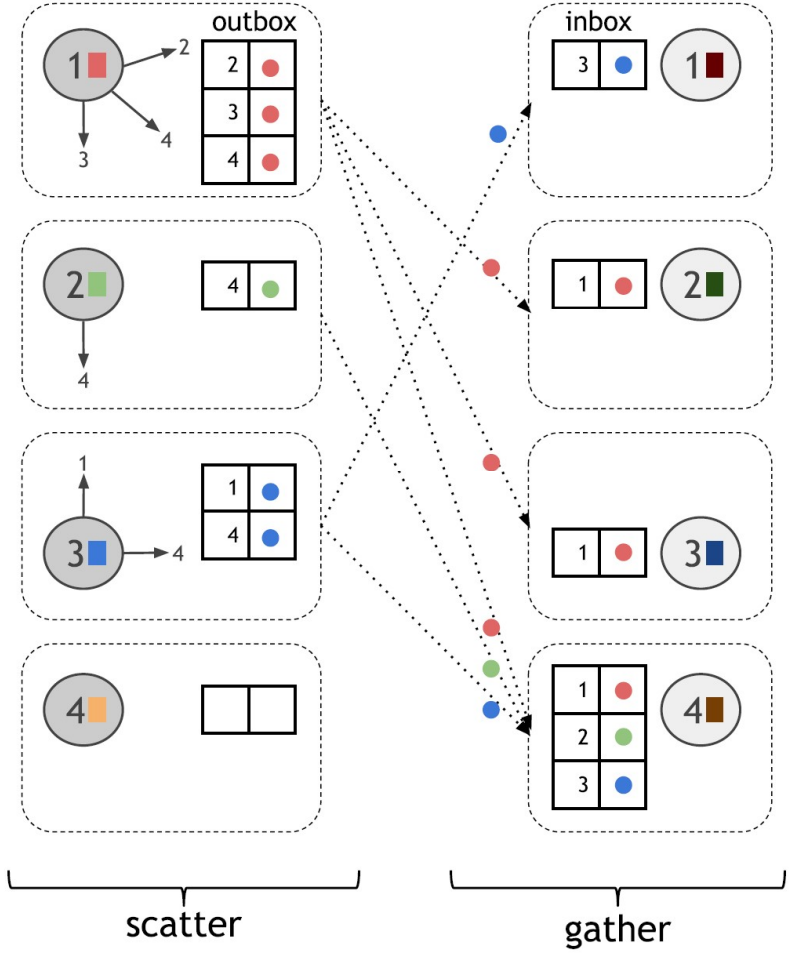
Table 1: A Review of Recent Graph Processing Frameworks on FPGAs

Works	Program. Model	Interconnect	PL[1]	App[2]	Graph Fmt.
Graphlily [18]	Linear Algebra	Crossbar	HLS	4	CPSR
GraphScale [12]	Vertex-Centric	Two-level Crossbar	HDL	3	Inverse CSR
ThunderGP [6, 7]	GAS	Shuffling Logic [4]	HLS	7	COO
ReGraph [5]	GAS	Multi-stage Butterfly	HLS	3	COO
ACTS [21]	GAS	*None*	HLS	4	ACTPACK
GraFlex	Scatter-Gather	Multi-stage Butterfly	HLS	6	CSR

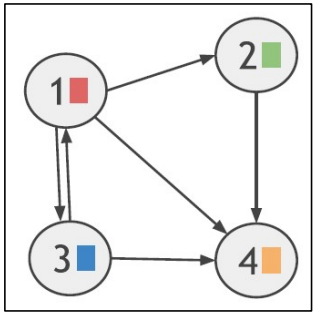
[1] Required Programming Language for development.
[2] Number of evaluated graph applications with the system.

- Remaining challenges to be tackled…
  - The linear algebra and the GAS (Gather-Scatter-Apply) model may limit the frameworks' **expressive efficiency** to certain graph applications. GraphScale is flexible (vertex-centric), but developed with HDL.
  - Prior works' customized crossbars and data shuffling logic are **resource-intensive** and require non-trivial effort to scale up.
  - Works based on **edge parallelism** impose much higher **storage requirements** ($2|E|$) than the **vertex-parallel** ones ($|V|+|E|$), limiting the input graph size.

Background: The Scatter-Gather Model



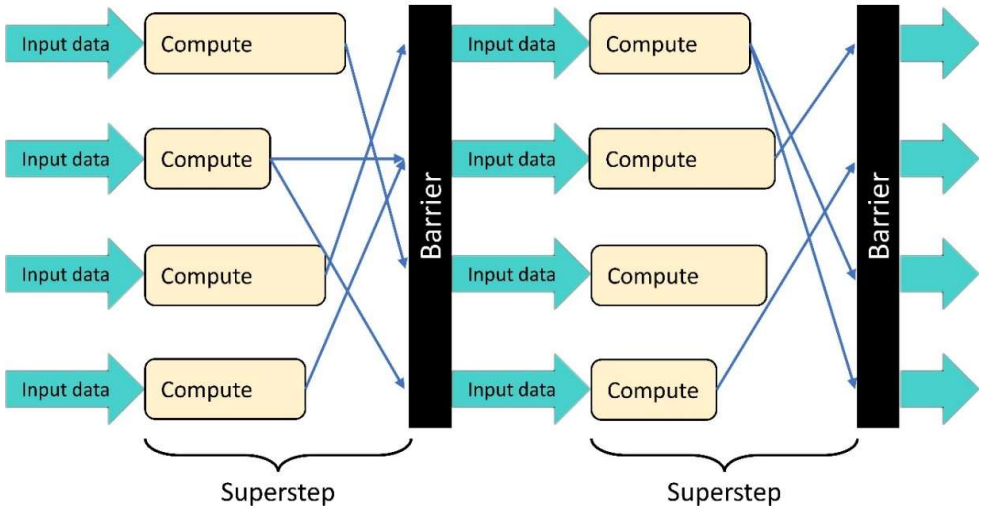
Example Graph



- Scatter-Gather Model: A vertex-parallel abstraction for distributed graph processing
  - Scatter PE
    - Sending out messages from active source vertices to their adjacent destinations
  - Gather PE
    - Collecting messages and updating vertex states accordingly

Image source: *Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2017. High-level programming abstractions for distributed graph processing. IEEE Transactions on Knowledge and Data Engineering 30, 2 (2017), 305–324.*

Background: The BSP Paradigm

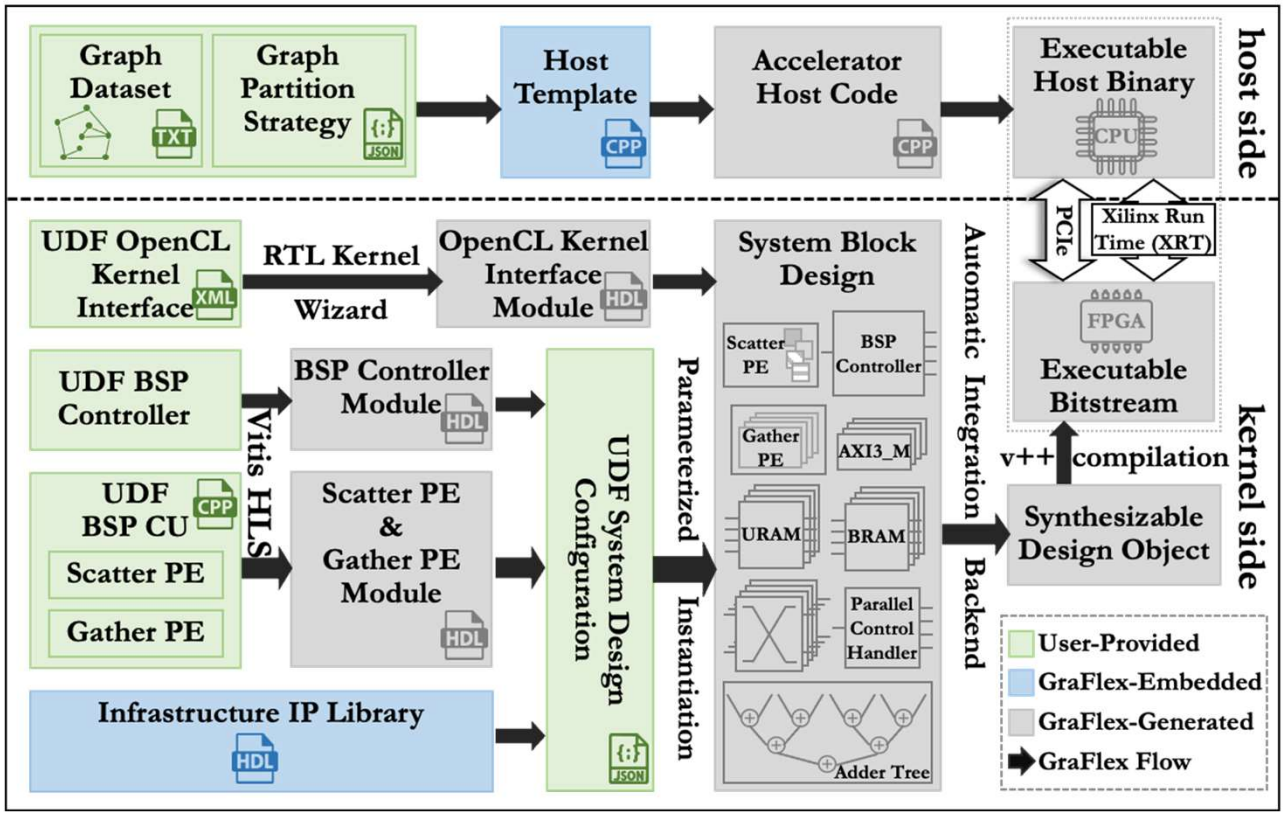


Algorithm 1: Scatter-Gather Model with BSP Paradigm

```
Input: Graph G = (V, E) partitioned as {G_i = (V_i, E_i)}
1  bspSuperstep = 0;
2  foreach partition G_i assigned to CU_i do
3      Initialize vertex state S_i;
4      Prepare activeVertices_i from V_i;
5      while activeVertices_i ≠ ∅ do
6          foreach vertex v_i ∈ activeVertices_i do
7              do in parallel
8                  outPacket_i ← Scatter(v_i);
9                  bspNetwork(outPacket_i, inPacket_j);
10                 S_j ← Gather(inPacket_j, S_j);
11         Update activeVertices_i;
12         bspSynchronize();
13         bspSuperstep = bspSuperstep + 1;
```

- The BSP parallel programming paradigm
  - **Parallel computing units (CUs)** for local computation
  - **Interconnection** network that allows data exchange among CUs
  - The central **controller** that enables **global synchronization** of multiple CUs
- Adopting BSP to the Scatter-Gather model in GraFlex
  - One Scatter PE and a group of Gather PEs are mapped to a BSP CU
  - Multiple CUs are managed by a central controller
  - Implementation of customized interconnection network

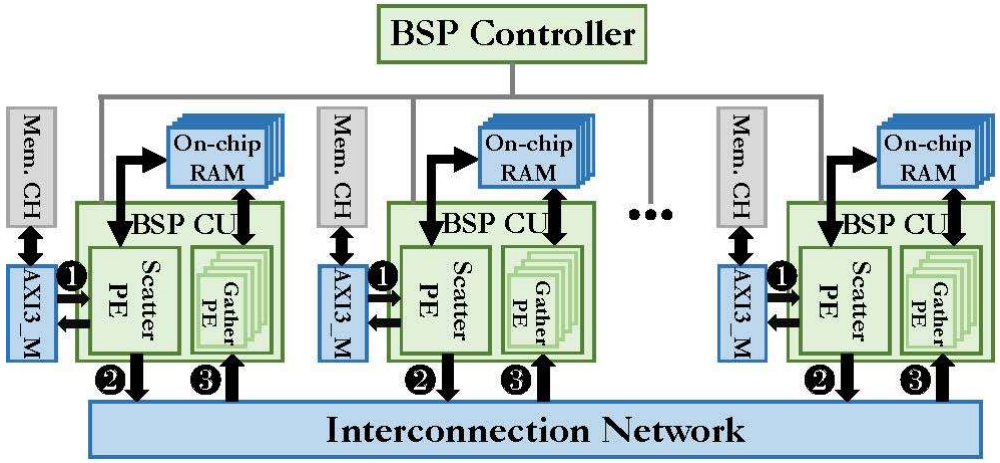
GraFlex Overview: Framework Flow



- Bottom Layer: RTL-based infrastructure IP library
- Middle Layer: Automatic GraFlex flow
- Top Layer: HLS-based design paradigm

GraFlex Overview: Architecture

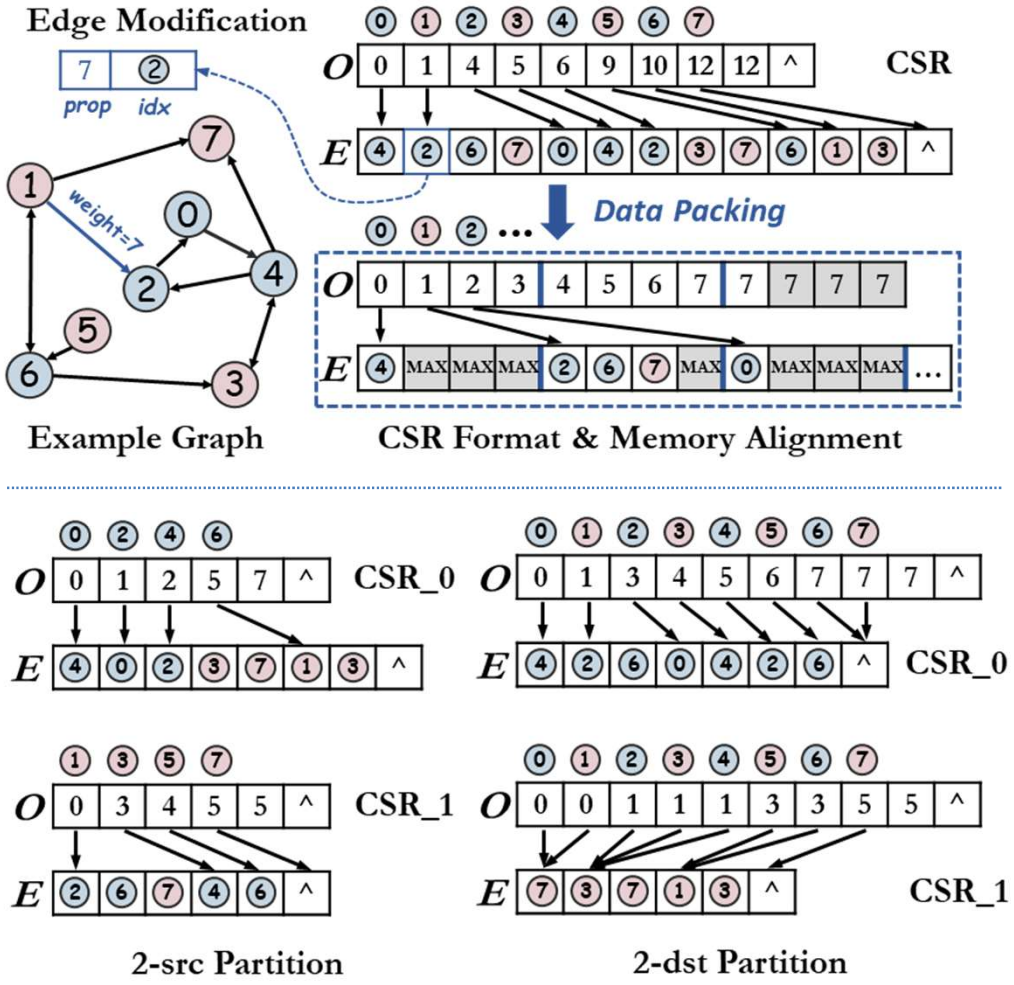
- Throughput-Matching Design: Balance between performance and resources



- ❶: Data transfer rate of the main memory: $\frac{W_{AXI\_D}}{W_e} \times N_C$
- ❷: Bandwidth of the interconnect: $N_{I/O}$
- ❸: Data processing rate of Gather PEs: $\frac{N_G}{II_{Gather}}$

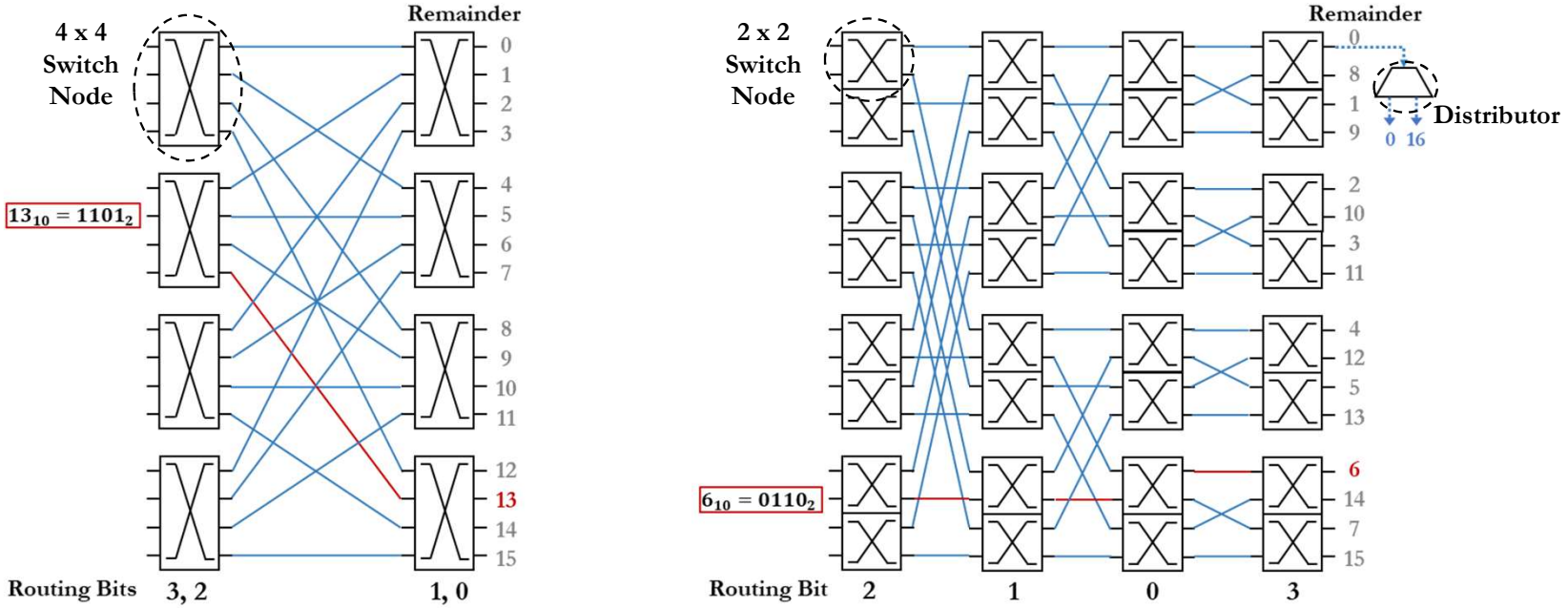
$$\frac{W_{AXI\_D}}{W_e} \times N_c \leq N_{I/O} \leq \frac{N_G}{II_{Gather}}$$

Graph Data Organization



- Compact Storage
  - CSR (|V|+|E|) over COO (2|E|)
- Pre-processing & Partitioning
  - m-src, n-dst 2D modulo partition
- Application-specific Modification
  - Carry property values in the high significance bits (prop, idx)
- Data Packing
  - Aligning with AXI DATA_WIDTH to fully utilize the memory bandwidth
  - Padding extra data as sentinel values

Customized Interconnection Network

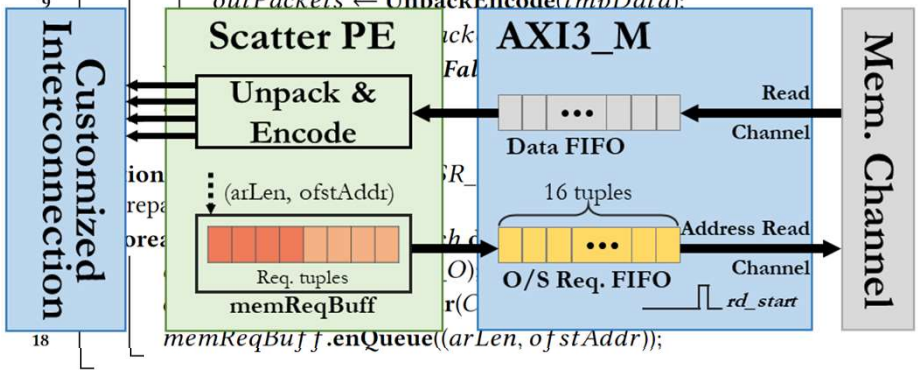


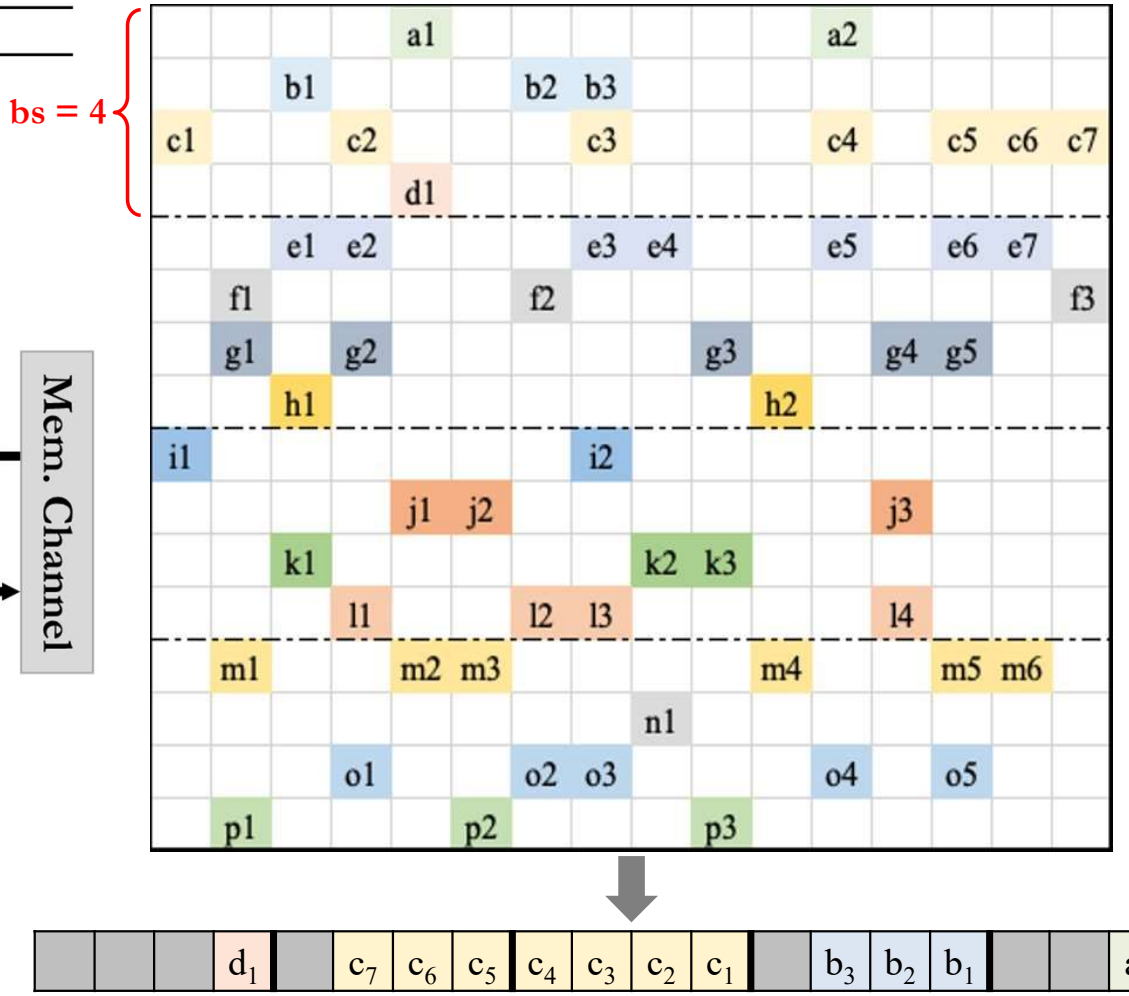
- Multi-stage butterfly topology
  - Reducing resource usage, mitigating routing and timing difficulties
- Network Customization
  - 2x2 or 4x4 switch units at each stage **(switch type <-> latency)**
  - Distributed RAM or BRAM for FIFO implementation **(depth <-> width <-> resources)**
  - Jointly consideration of memory interfaces and PE throughput **(recall throughput-matching design)**

Memory Access Optimizations

Listing 1: Coalesced Memory Access

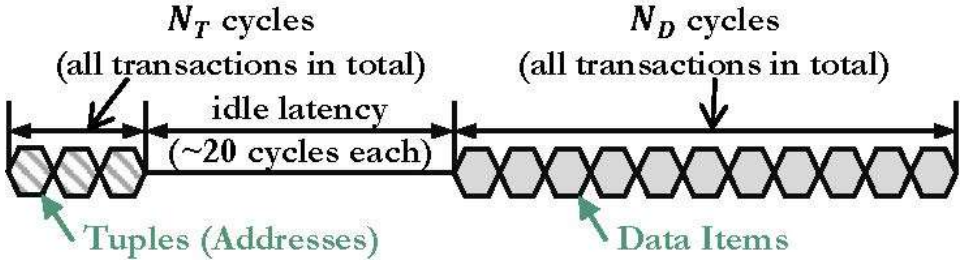
```
1  Function feedNetwork(rdPort, netPort, memReqBuff):
2      for k from 1 to ⌈memReqBuff.size() / 16⌉ do
3          Prepare tupleBatch from memReqBuff;
4          foreach reqTuple ∈ tupleBatch do
5              rdPort.writeReq(reqTuple);
6          rdPort.rd_start ← True;
7          do
8              tmpData ← rdPort.read();
9              outPackets ← UnpackEncode(tmpData);
18         memReqBuff.enQueue((arLen, ofstAddr));

19 Function scatterMain(CSR_O, CSR_E, rdPort, netPort):
20     Initialize memReqBuff1 and memReqBuff2;
21     for n from 1 to N_batch do
22         do in parallel
23             feedNetwork(rdPort, netPort, memReqBuff1);
24             genMemTuple(CSR_O, CSR_E, memReqBuff2);
25         Swap memReqBuff1 and memReqBuff2;
```





Memory Access Optimizations

- Coalesced Memory Access Modeling



$$N_D = \sum_{v=0}^{batchSize} \left\lceil \frac{out\_degree(v)}{W_{AXI\_D}/W_e} \right\rceil, N_T = \left\lceil \frac{N_D}{16} \right\rceil$$

$$BW_{ori} = \frac{N_D}{N_{Idle} \times batchSize + N_T + N_D}$$

$$BW_{opt} = \frac{N_D}{N_{Idle} \times \left\lceil \frac{N_T}{16} \right\rceil + N_T + N_D}$$

Table 7: Mean and Variance of Memory Bandwidth Utilization

Graph	MG	HW	OR	PK	AM	TC	GG	HD
$\overline{BW_{ori}}$	83.67%	40.19%	43.26%	16.77%	7.85%	12.89%	6.63%	8.22%
$\overline{BW_{opt}}$	87.38%	83.94%	84.57%	70.50%	66.51%	66.78%	64.27%	66.53%
$\overline{BW_{opt}}/\overline{BW_{ori}}$	1.04×	2.09×	1.95×	4.20×	8.47×	5.18×	9.69×	8.09×

- **1.04x ~ 9.69x** improvement in memory bandwidth utilization
- Generally, more benefits are obtained from sparser graphs

Evaluation

- Experimental Setup
  - System settings
    - AMD Ryzen 5900X CPU + 128 GB DDR4 Memory
    - AMD Xilinx Alveo U280 Board (two-stack 8 GB HBM2)
  - Graph Datasets
    - Eight real-world graphs publicly available
  - Benchmarking applications
    - Six popular applications
    - BFS, SSSP, WCC, CC, SpMV and PageRank
  - Baseline
    - ThunderGP Alveo U280 implementation*
- System Configurations per Application
  - Graph Partition Scheme
  - CSR Data Format
  - Interconnect Configurations
  - Initiation Interval (II) for the Gather PEs

Table 2: Graph Datasets for Evaluation

Graphs	Type	$\|V\|$	$\|E\|$	Density
mouse-gene (MG)	Undirected	45.1K	14.5M	7.13 E−3
ca-hollywood-2009 (HW)	Undirected	1.1M	56.3M	4.65 E−5
com-orkut (OR)	Undirected	3.1M	117.2M	1.22 E−5
pokec-relationships (PK)	Directed	1.6M	30.6M	1.20 E−5
amazon-2008 (AM)	Directed	735.3K	5.2M	9.62 E−6
wiki-topcats (TC)	Directed	1.8M	28.5M	8.80 E−6
web-google (GG)	Directed	875.7K	5.1M	6.65 E−6
web-hudong (HD)	Directed	2.0M	14.9M	3.73 E−6

Table 3: System Configurations per Application

App.	Partition	W_{AXI_D}	$W_{prop} + W_{idx}$ [1]	$(N_{net}, N_{I/O}, W_{pkt})$ [2]	II_{Gather}
BFS	32-src	128b	0b + 32b	(1, 128, 32b)	2
SSSP	32-src	128b	8b + 24b	(1, 128, 48b)	2
WCC	32-src	128b	0b + 32b	(1, 128, 48b)	2
CC	32-src	128b	0b + 32b	(1, 128, 32b)	2
SpMV	4-src, 8-dst	256b	32b + 32b	(8, 16, 66b)	3
PR	4-src, 8-dst	256b	32b + 32b	(8, 16, 66b)	3

[1] W_{prop}: bit width of the property; W_{idx}: bit width of the index
[2] N_{net}: number of networks, W_{pkt}: bit width of the network packets

*Xinyu Chen, Feng Cheng, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2022. ThunderGP: resource-efficient graph processing framework on FPGAs with HLS. ACM Transactions on Reconfigurable Technology and Systems 15, 4 (2022), 1–31.

Evaluation

Table 4: Traversal Throughput (MTEPS)

Best Case Worst Case

App.	BFS	SSSP	WCC	CC	SpMV	PR
MG	8,731	8,459	11,673	8,477	12,310	11,240
HW	11,013	11,060	13,278	10,681	12,018	9,260
OR	9,876	10,137	11,881	9,574	-	-
PK	7,824	7,916	10,202	7,542	6,266	5,863
AM	5,048	5,126	6,909	4,902	2,535	2,420
TC	3,376	3,410	6,072	3,278	4,519	4,302
GG	4,429	4,511	5,805	4,275	2,117	2,014
HD	5,160	5,265	6,890	5,005	2,658	2,382
Geo. Mean[1]	**6,039**	**6,081**	**8,273**	**5,851**	**4,812**	**4,371**
ThunderGP [6]	4,251	3,689	3,959	3,808	5,308	3,878
Speedup	**1.42×**	**1.65×**	**2.09×**	**1.54×**	**0.91×**	**1.13×**

[1]Excluding OR dataset for fair comparison with ThunderGP [6].

- Throughput Results
  - Generally, throughput goes down with graph density (same app.), which entails that the **upstream data transfer** (❶) becomes the **performance bottleneck.**
  - GraFlex performs best on **WCC**, achieving **2.09x** speedup in geometric mean
  - For **BFS**, **SSSP** and **CC**, GraFlex also achieves significant speedup on average
  - For **SpMV**, GraFlex maintains high performance for denser graph datasets (MG & HW).
  - Sparser datasets bring more severe performance degradation due to inefficient global reduction of partial sums.

Evaluation

Table 5: Resource Utilization

App.	BRAM		URAM		CLB		DSP	
	G	T	G	T	G	T	G	T
BFS	26.79% (6.34%)	61%	66.67%	60%	55.62% (12.96%)	94%	0.04%	0.04%
SSSP	39.48% (12.68%)	65%	66.67%	53%	57.68% (17.95%)	88%	0.04%	0.04%
WCC	33.13% (12.68%)	62%	93.33%	62%	63.31% (17.85%)	90%	0.04%	0.04%
CC	23.61% (6.34%)	62%	66.67%	60%	57.19% (13.06%)	91%	0.04%	0.04%
SpMV	71.23% (0%)	66%	43.33%	68%	77.59% (14.12%)	98%	22.74%	2.44%
PR	71.23% (0%)	63%	43.33%	60%	76.75% (14.64%)	92%	22.74%	0.4%

Note : **G**: GraFlex, with interconnect resource breakdown in brackets (w.r.t hardware budget, if any); **T**: ThunderGP [6].

Table 6: Kernel Frequency and Power Consumption

	BFS		SSSP		WCC		CC		SpMV		PR	
	G	T	G	T	G	T	G	T	G	T	G	T
Freq. (MHz)	175	250	180	243	160	258	170	251	190	250	180	267
Power (W)	42	52	44	52	43	49	42	48	51	58	50	50

Note : **G**: GraFlex (Ours); **T**: ThunderGP [6].

- Resource, Frequency and Power
  - Compared to ThunderGP, GraFlex achieves **~27% CLB saving** on average due to the resource-efficient interconnection network
  - GraFlex achieves 160~190 MHz among all six applications implemented. Further frequency improvement requires systematic physical design optimizations
  - Compared to ThunderGP, GraFlex achieves a **~12% average reduction** in terms of power consumption.

Evaluation

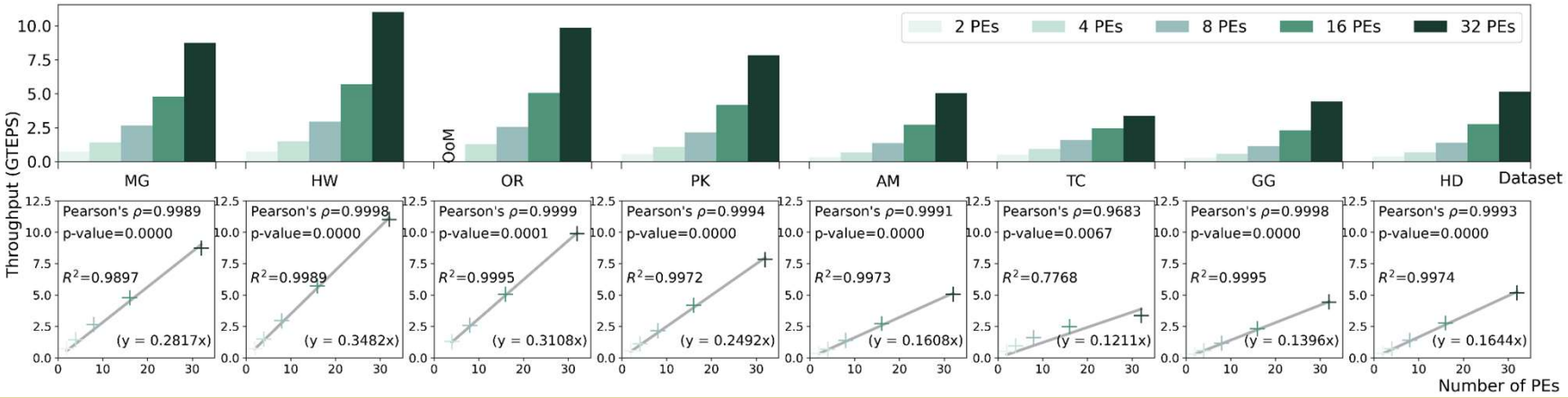
- Traversal Throughput versus Algorithm Throughput
  - Example: avoiding redundant traversals with more efficient BFS algorithm

Table 8: Algorithm Throughput of BFS with Two Implementations

Throughput (MTEPS)	MG	HW	OR	PK	AM	TC	GG	HD
Bellman-Ford[1] (T_B)	1,449	1,375	1,235	1,565	240	25	261	369
Dijkstra's[2] (T_D)	7,689	8,180	7,624	5,528	1,377	1,166	1,349	1,369
Speedup (T_D/T_B)	5.31	5.95	6.17	3.53	5.74	46.64	5.17	3.71

[1]Kernel frequency = 175 MHz. [2]Kernel frequency = 160 MHz.

- Scalability Evaluation
  - BFS Traversal Throughput on Different Datasets
  - Almost-Linearity of Throughput *v.s.* the number of PEs



Thanks for your attention!

Q & A