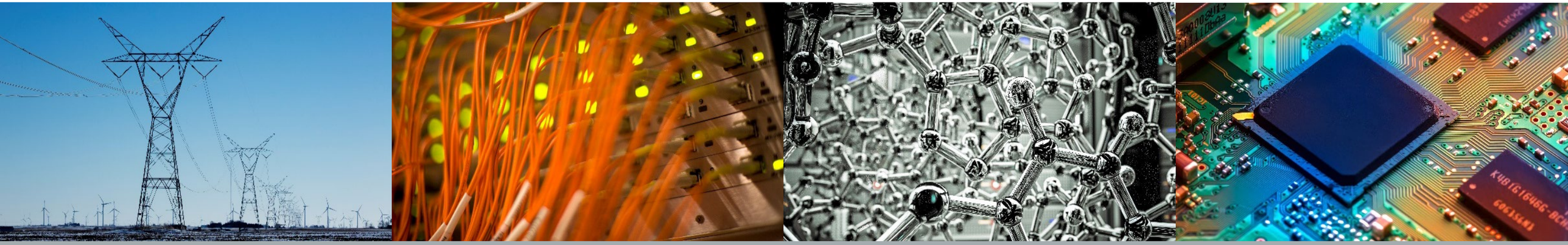
EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs

Seung Won Min[1], Vikram Sharma Mailthody[1], Zaid Qureshi[1], Jinjun Xiong[2], Eiman Ebrahimi[3], Wen-mei Hwu[13]

[1]University of Illinois at Urbana-Champaign, [2]IBM Research, [3]NVIDIA Research

August 18th, 2021



ILLINOIS
Electrical & Computer Engineering
GRAINGER COLLEGE OF ENGINEERING

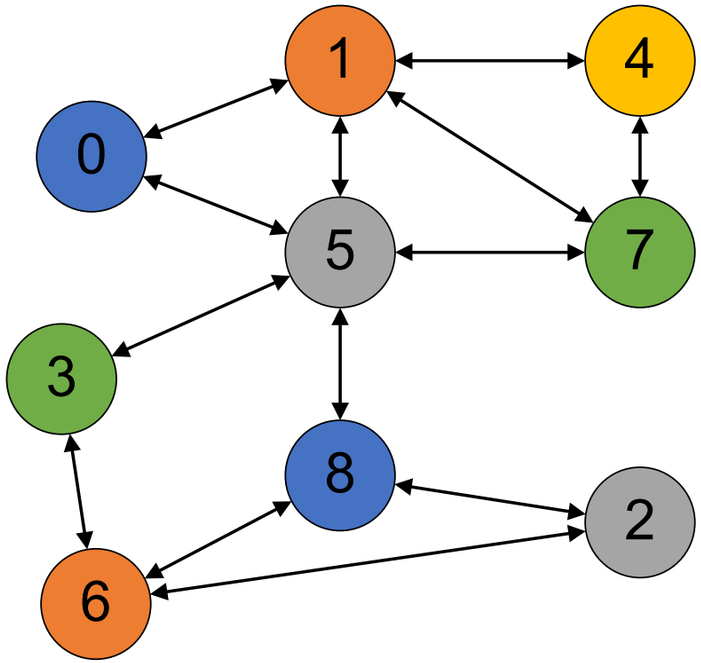
C3SR center for cognitive computing systems research

The IMPACT Research Group

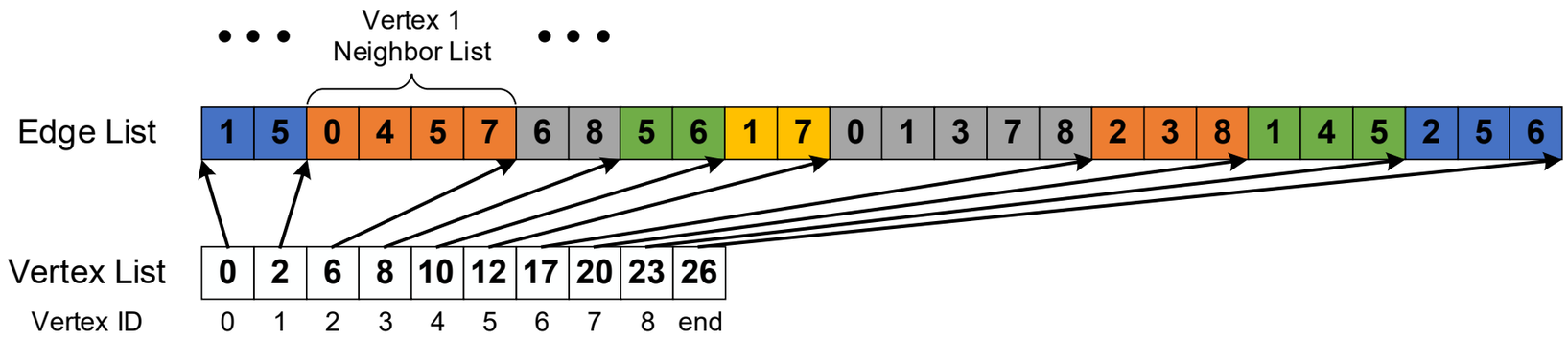
VLDB 2021

Graph traversal basics

- Short summary: Start from certain vertices (or a vertex) and keep traversing neighboring vertices in an iterative manner



Graph
(Logical view)



CSR
(Store format in computer memory)

Graph traversal with GPUs

- Opportunity: Massive number of vertices to traverse and identical workloads per traversal
  - Ex) cuGraph, Gunrock, etc.

Algorithm 1: High-level Graph Traversal Flow

```
set_initial_active_vertex()
while there exist active vertices in G do
    for all vertices v1 in Graph G do
        if v1 is active then
            set v1 as inactive
            for all neighbors v2 of v1 do
                application_dependant_workload()
                if application_dependant_condition() then
                    set v2 as active
                end
            end
        end
    end
end
```

Motivation of this work

- GPU memory might be too small for some graphs!
  - Graphs should be stored somewhere else and accessed in runtime
  - Irregular data access pattern in CSR causes a poor spatial data locality, and it makes the block data transfer method inefficient

- Related works:
  - HALO (VLDB'20): CSR reordering for higher data spatial locality
  - Subway (EuroSys'20): Runtime data partitioning and data transfer
  - Limitations: Still under-utilized PCIe bandwidths

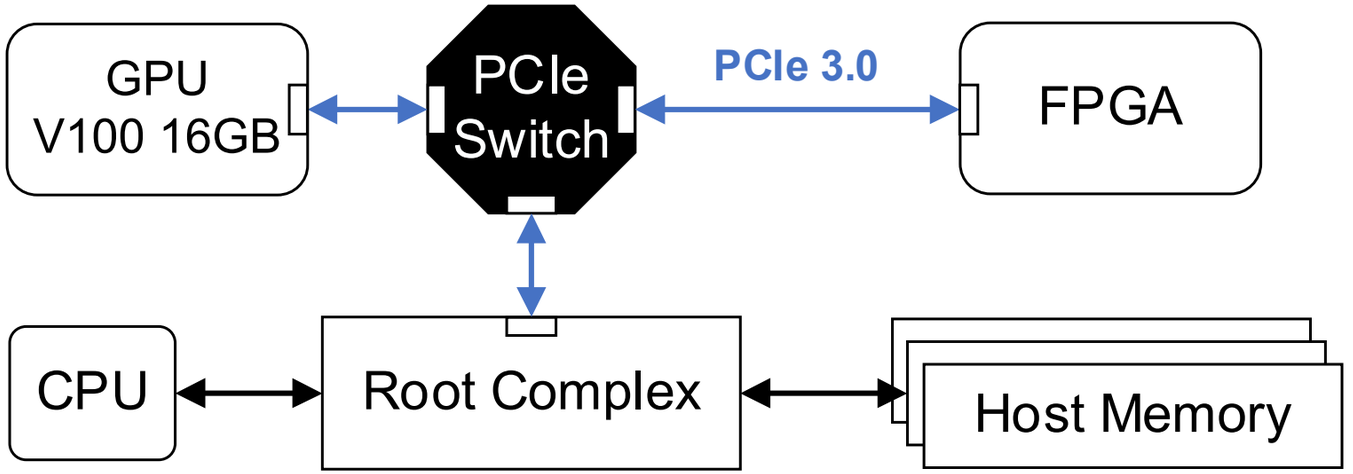
Forget about data copy, just access it!

- Modern GPUs can directly access the CPU memory, just like its own memory
  - Also referred as *Zero-copy*
  - Accesses are done in a *cacheline* granularity

- Question) But can we saturate PCIe only with zero-copy accesses?

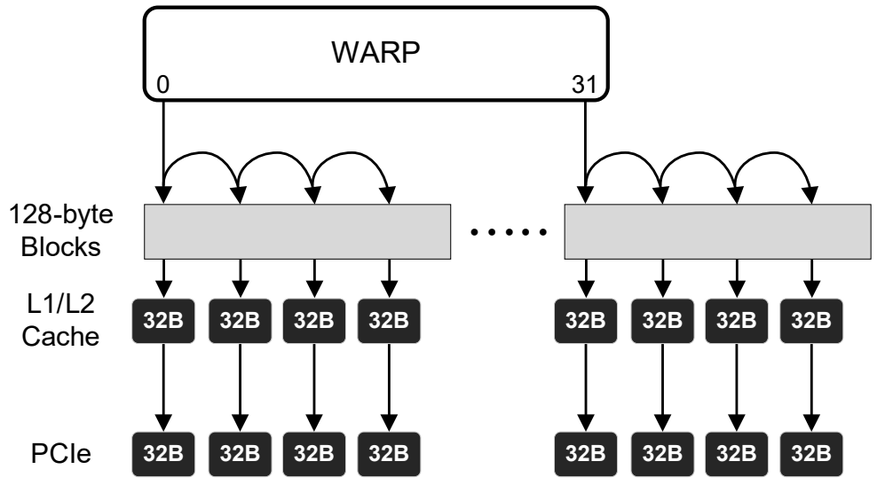
Understanding PCIe traffic with Zero-copy

- System setup



- Toy example used
  - Perform a sequential Zero-copy memory read from CPU memory using GPU kernel

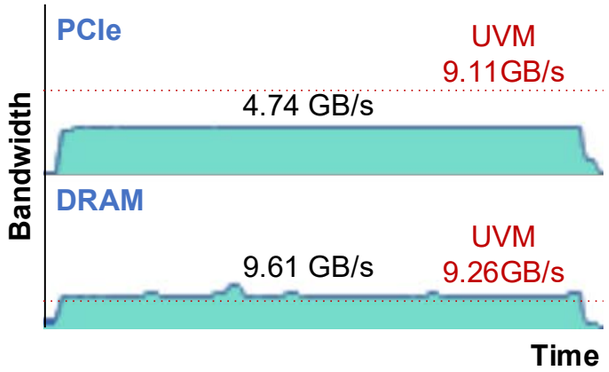
Toy example & Measured PCIe bandwidth



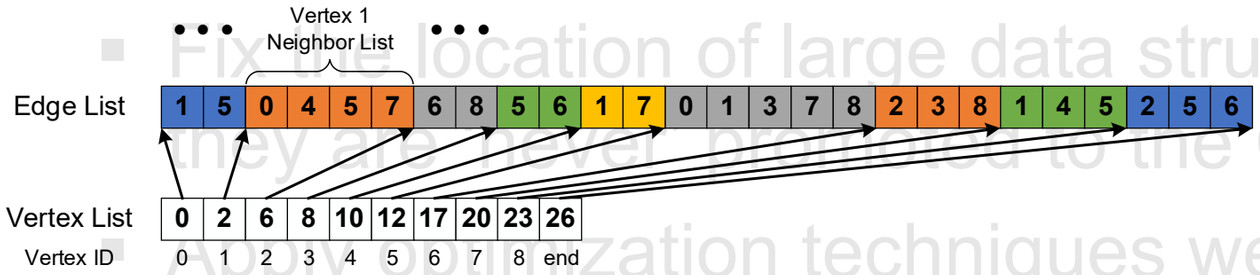
(a) Strided

Measured Peak PCIe 3.0 Bandwidth: 12.8GB/s

(a) Strided



EMOGI: Zero-copy based graph traversal



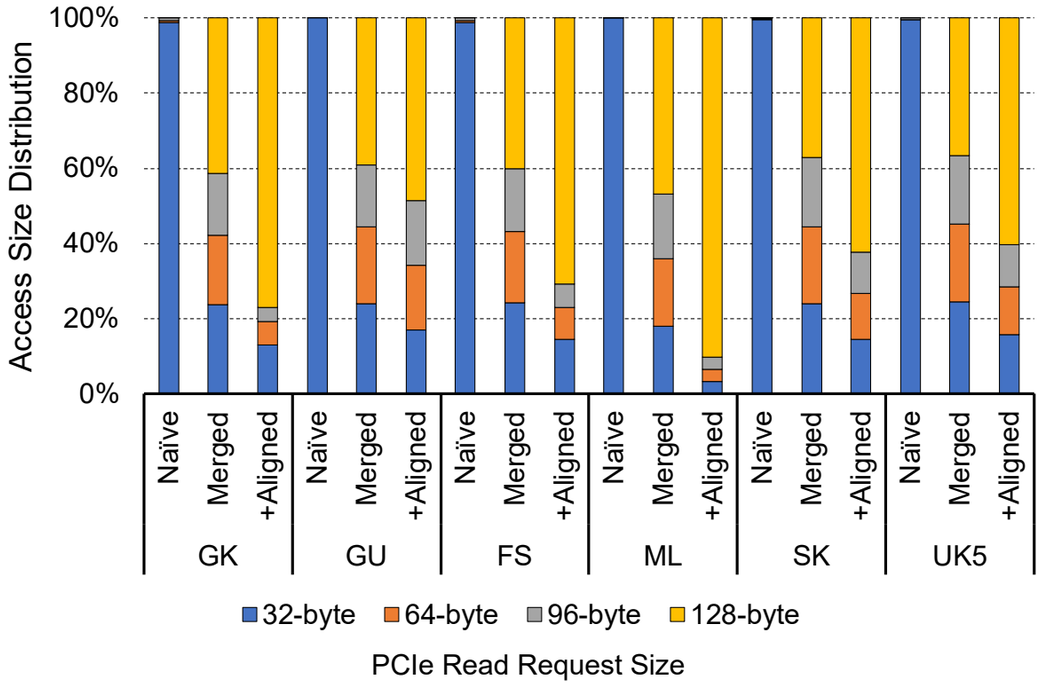
Listing 1: Uncoalesced Memory Access

```
1  void naive(*edgeList, *offset, ...) {
2      thread_id = get_thread_id();
3      ...
4      start = offset[thread_id];
5      end = offset[thread_id + 1];
6
7      // Each thread loops over a chunk of edge list
8      for (i = start; i < end; i++) {
9          edgeDst = edgeList[i];
10         ...
11     }
12     ...
13 }
```

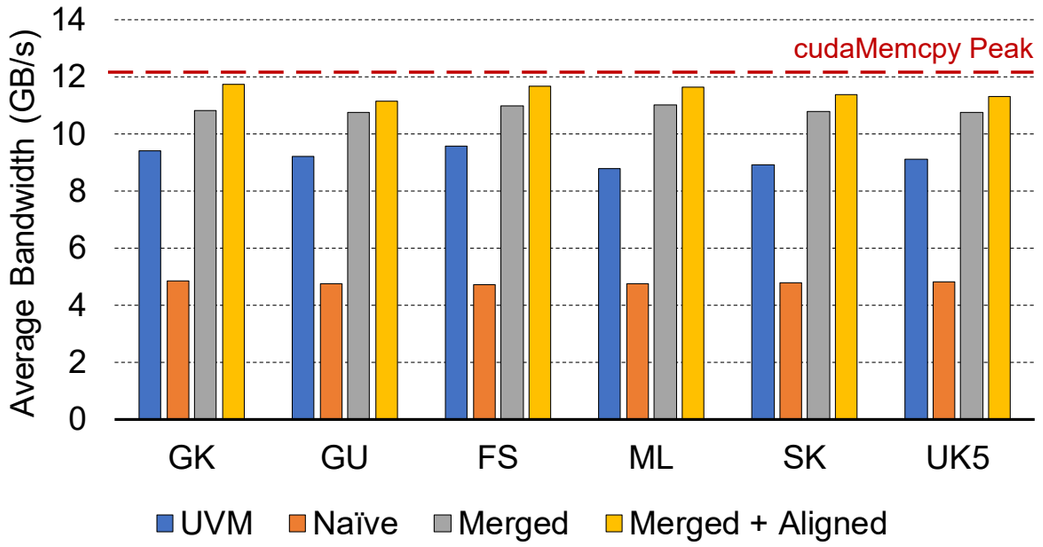
Listing 2: Coalesced Memory Access (Merged + Aligned)

```
1  #define WARP_SIZE 32
2
3  void aligned(*edgeList, *offset, ...) {
4      thread_id = get_thread_id();
5      lane_id = thread_id % WARP_SIZE;
6      // Group by warp
7      warp_id = thread_id / WARP_SIZE;
8      ...
9      start_org = offset[warp_id];
10     // Align starting index to 128-byte boundary
11     start = start_org & ~0xF; // 8-byte data type
12     end = offset[warp_id + 1];
13
14     // Every thread in a warp goes to the same edgelist
15     for (i = start; i < end; i += WARP_SIZE) {
16         // Prevent underflowed accesses
17         if (i >= start_org) {
18             edgeDst = edgeList[i + lane_id];
19             ...
20         }
21     }
22     ...
23 }
```

EMOGI: BFS case-study



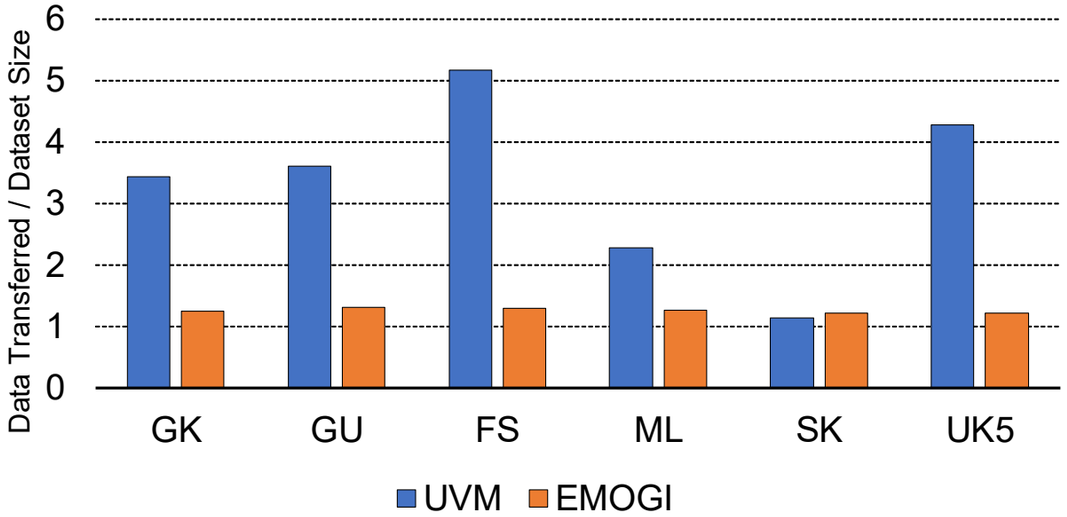
PCIe Request Size Distribution



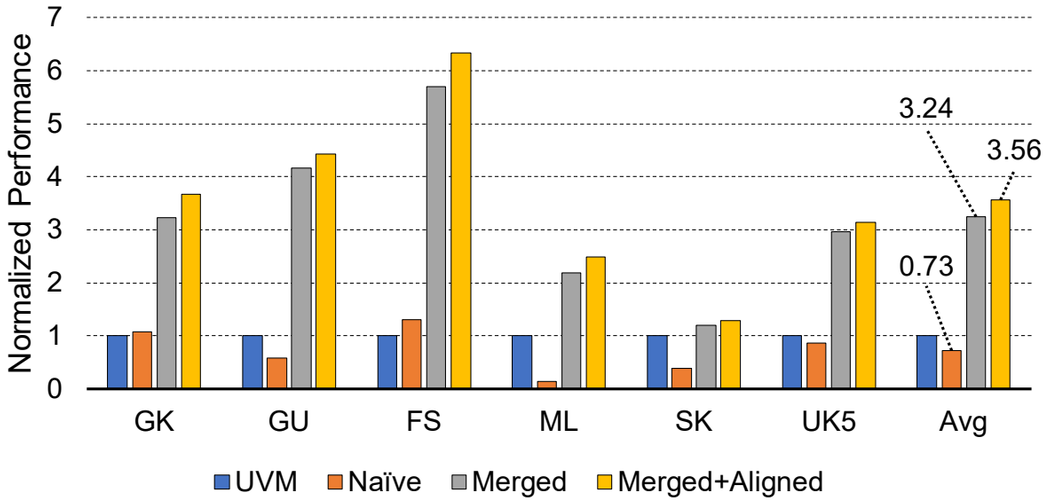
PCIe Bandwidth

Sym.	Graph	Number		Size (GB)	
		$\|V\|$	$\|E\|$	$\|E\|$	$\|w\|$
GK	GAP-kron [10]	134.2M	4.22B	31.5	15.7
GU	GAP-urand [10]	134.2M	4.29B	32.0	16.0
FS	Friendster [52]	65.6M	3.61B	26.9	13.5
ML	MOLIERE_2016 [48]	30.2M	6.67B	49.7	24.8
SK	sk-2005 [12–14]	50.6M	1.95B	14.5	7.3
UK5	uk-2007-05 [13, 14]	105.9M	3.74B	27.8	13.9

EMOGI: BFS case-study

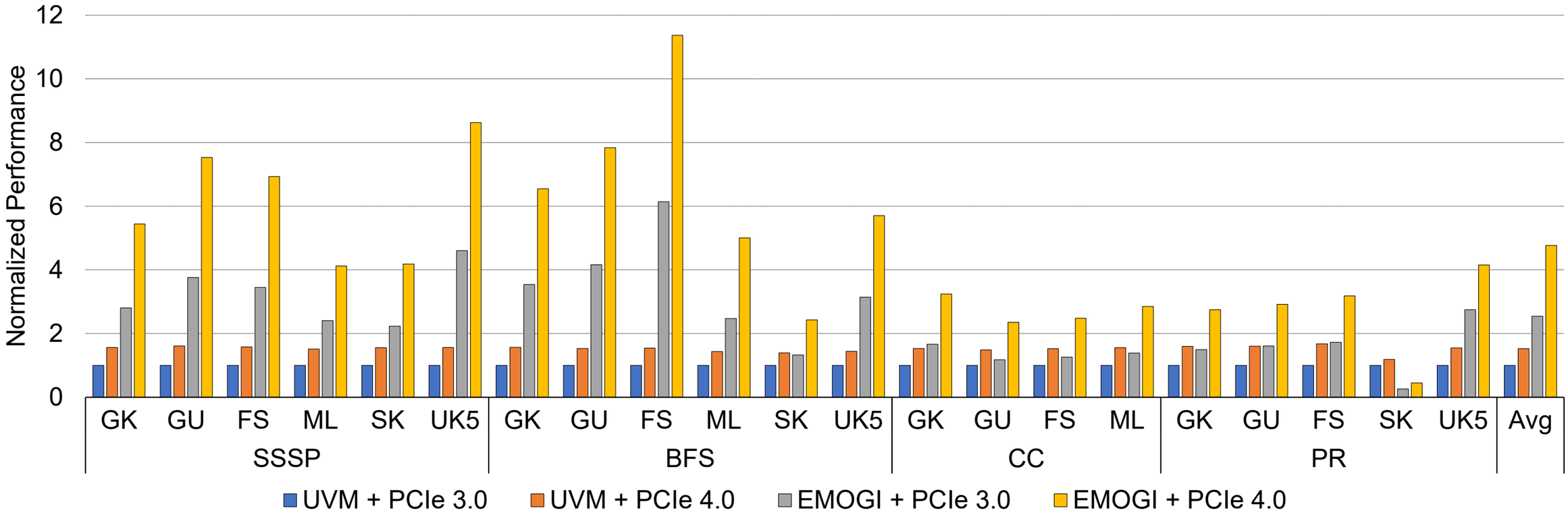


I/O Read Amplification



BFS Execution Time

EMOGI: Overall performance comparison



UVM vs. EMOGI
(A100 + PCIe 3.0 / 4.0)

EMOGI: Comparison with previous works

HALO (VLDB'20) vs. EMOGI
(Titan Xp + PCIe 3.0)

Application	Graph	Exe. Time		Speedup
		HALO	EMOGI	
BFS	ML	9.54s	4.43s	2.15×
	FS	8.27s	2.59s	3.19×
	SK	2.17s	1.62s	1.34×
	UK5	6.03s	4.00s	1.51×

Subway (EuroSys'20) vs. EMOGI
(Tesla V100 + PCIe 3.0)

Application	Graph	Exe. Time		Speedup
		Subway	EMOGI	
SSSP	GK	20.96s	7.94s	2.64×
	FS	14.95s	6.97s	2.14×
	SK	8.99s	3.92s	2.30×
	UK5	25.78s	8.08s	3.19×
BFS	GK	6.88s	1.66s	4.14×
	FS	4.22s	1.49s	2.83×
	SK	1.69s	0.85s	1.99×
	UK5	8.75s	1.85s	4.73×
CC	GK	6.34s	3.11s	2.04×
	FS	4.31s	2.75s	1.57×

Thank You!

Email: min16@illinois.edu
Github: https://github.com/illinois-impact/EMOGI

ILLINOIS
Electrical & Computer Engineering
GRAINGER COLLEGE OF ENGINEERING