Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven Differential Scheduling

Long Zheng[1], Xianliang Li[1], Yaohui Zheng[1], **Yu Huang**[1], Xiaofei Liao[1], Hai Jin[1], Jingling Xue[1] Zhiyuan Shao[1], and Qiang-Sheng Hua[1]

[1]Huazhong University of Science and Technology
[2]University of New South Wales

July 15-17, 2020

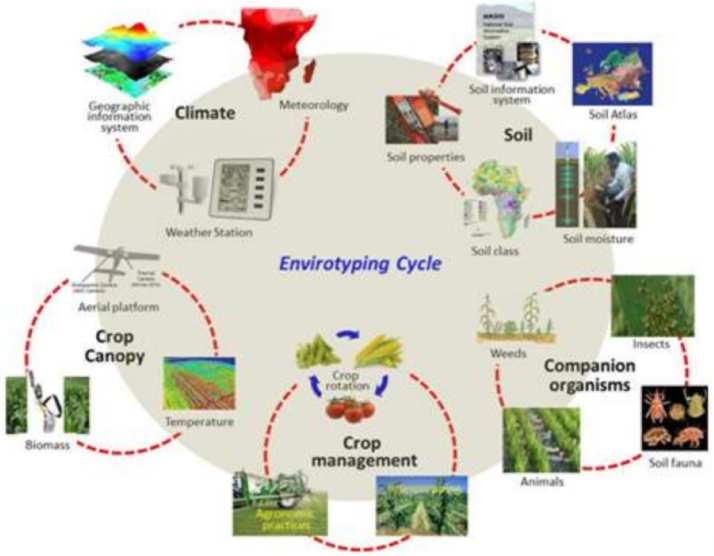
華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

UNSW
THE UNIVERSITY OF NEW SOUTH WALES

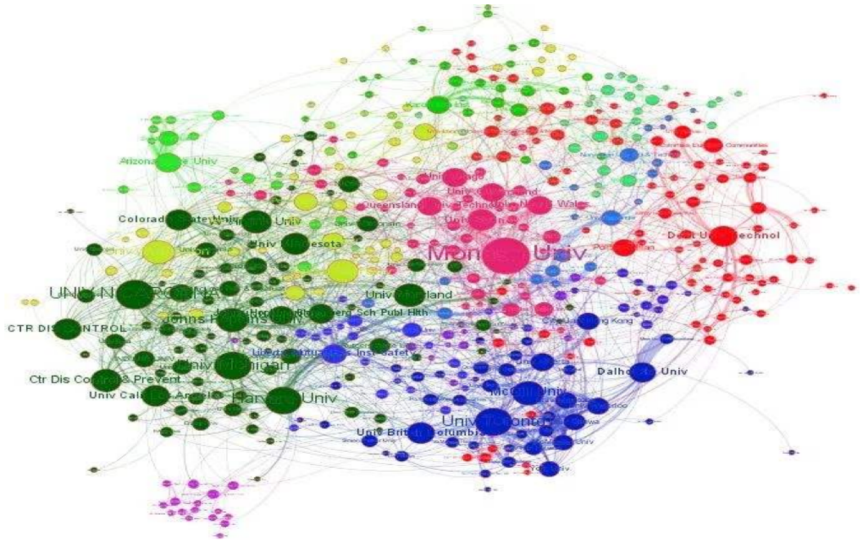
Graph Processing Is Ubiquitous

Relationship Prediction

Recommendation Systems



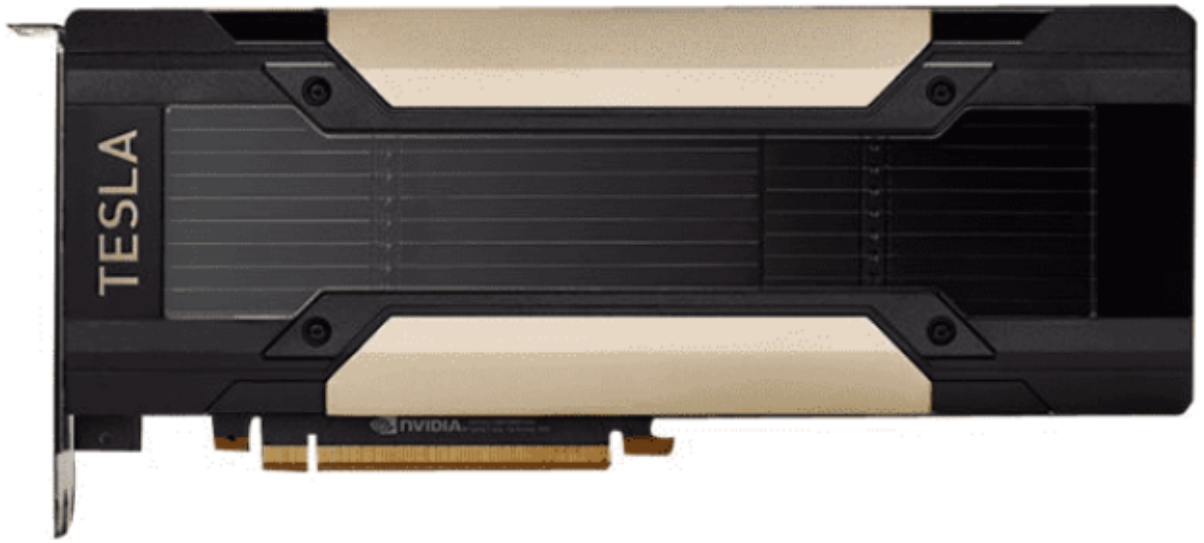
Information Tracking



Knowledge Mining

Graph Processing: CPU vs. GPU

GPU V100



Performance	Double-Precision: 7.8TFLOPS, Single-Precision: 15.7TFLOPS
InterConnect Bandwidth	NVLINK 300GB/s
Memory	32GB HBM2, 1134GB/s

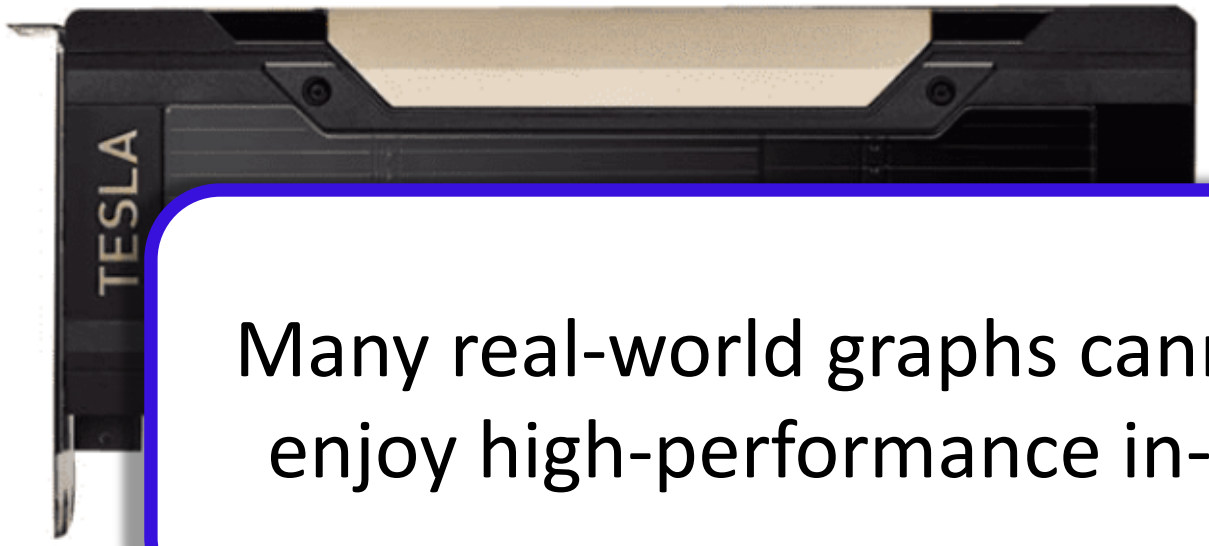
Data source: V100 Performance, https://developer.nvidia.com/hpc-application-performance

Application	Metric	Test Modules	Bigger is better	Dual Cascade Lake 6240 (CPU-Only)	1x V100 32GB SXM2
AMBER [PME-Cellulose_NPT_4fs]	ns/day	DC-Cellulose_NPT	yes	4.73	88.29
AMBER [PME-Cellulose_NPT_4fs]	NRF	DC-Cellulose_NPT	yes	1x	19x
AMBER [DC-Cellulose_NVE]	ns/day	PME-Cellulose_NVE	yes	4.73	100
AMBER [DC-Cellulose_NVE]	NRF	PME-Cellulose_NVE	yes	1x	21x
AMBER [DC-FactorIX_NPT]	ns/day	FactorIX (NPT)	yes	22.88	400
AMBER [DC-FactorIX_NPT]	NRF	FactorIX (NPT)	yes	1x	17x
AMBER [DC-FactorIX_NVE]	ns/day	FactorIX (NVE)	yes	23.41	454

GPU often offers 10x at least speedup over CPU for graph processing

Graph Processing: CPU vs. GPU

GPU V100



Performance	Double-Precision: 7.8TFLOPS, Single-Precision: 15.7TFLOPS
InterConnect Bandwidth	NVLINK 300GB/s
Memory	32GB HBM2, 1134GB/s

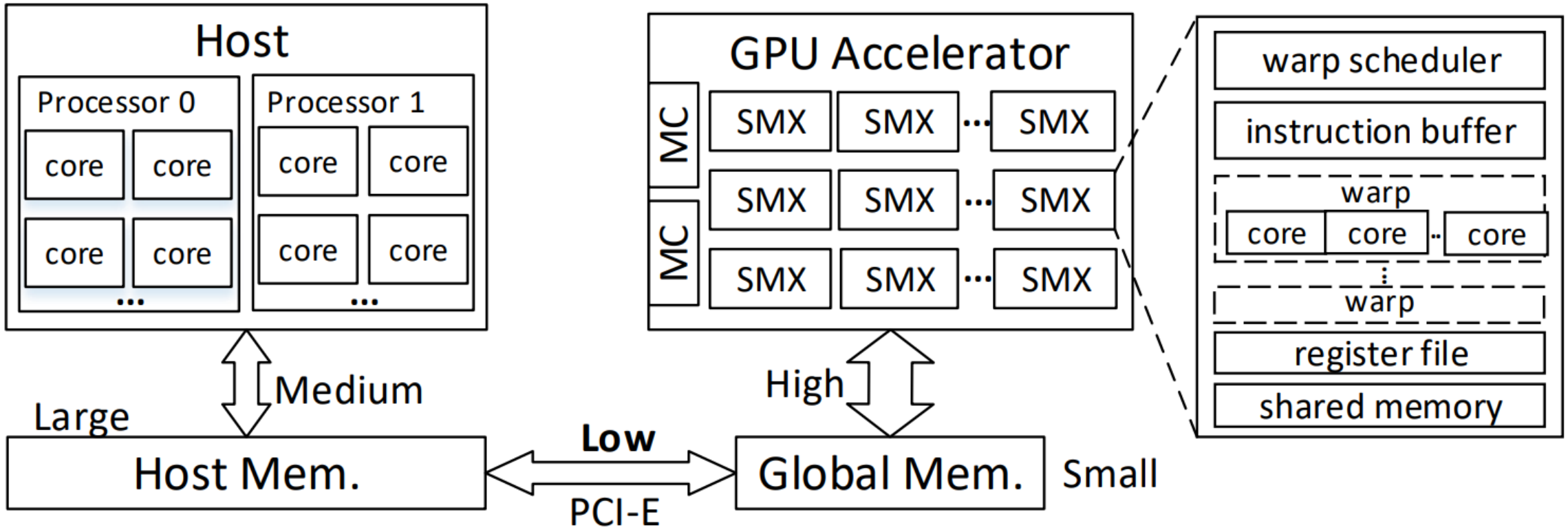
Many real-world graphs cannot fit into GPU memory to enjoy high-performance in-memory graph processing

Data source: V100 Performance, https://developer.nvidia.com/hpc-application-performance

Application	Metric	Test Modules	Bigger is better	Dual Cascade Lake 6240 (CPU-Only)	1x V100 32GB SXM2
AMBER [PME-Cellulose_NPT_4fs]	ns/day	DC-Cellulose_NPT	yes	4.73	88.29
AMBER [PME-	NRF	DC-	yes	1x	19x
					100
					21x
					400
AMBER [DC-FactorIX_NPT]	NRF	FactorIX (NPT)	yes	1x	17x
AMBER [DC-FactorIX_NVE]	ns/day	FactorIX (NVE)	yes	23.41	454

GPU often offers 10x at least speedup over CPU for graph processing

GPU-Accelerated Heterogeneous Architecture



The significant performance gap between CPU and GPU may severely limit the performance potential expected on the GPU-accelerated heterogeneous architecture.

Existing Solutions on GPU-Accelerated Heterogeneous Architecture

- Totem (PACT'12)
  - Partitioned into two large subgraphs, one for CPU, one for GPU
  - Significant load unbalance
- Graphie (PACT'17)
  - Subgraphs are partitioned and streamed to GPU
  - All subgraphs are transferred in their entirety
  - Bandwidth is wasted
- Garaph (USENIX ATC'17)
  - All the subgraphs are processed on CPU if the active vertices in the entire graph have a lot of (50%) outgoing edges
  - Processed on the host otherwise

A Generic Example of Graph Processing Engine

```
1  Procedure SimpleSubgraphEngine(Graph G)
2      Load G̃'s subgraphs in {G̃_1, ··· , G̃_n} into the host
3      VertexInitialization(G)
4      G̃_active ← FindActiveSubgraph(G)
5      while G̃_active ≠ ∅ do
6          foreach G̃_i ∈ G̃_active do
7              stream ← DispatchStream(G̃_i)
8              if G̃_i is not resident in GPU memory then
9                  GBuf ← AllocateDeviceMemory( )
10                 TransferData(stream, GBuf, G̃_i, CPU2GPU)
11             Kernel (stream, G̃_i)
12         G̃_active ← FindActiveSubgraph(G)

   /* Graph Processing Kernel on the GPU */
13 Procedure Kernel(Subgraph G̃)
14     foreach v ∈ G̃.SetOfVertices do
15         if v is active then
16             foreach e ∈ v.outedges do
17                 if Update(v, e) = SUCCESS then
18                     Activate(e.destination_vertex)
```

Vertices reside in GPU memory

Edges are streamed to GPU on demand

A graph is partitioned into many slices

A Generic Example of Graph Processing Engine

```
1  Procedure SimpleSubgraphEngine(Graph G)
2      Load G̃'s subgraphs in {G̃_1, ··· , G̃_n} into the host
3      VertexInitialization(G)
4      G̃_active ← FindActiveSubgraph(G)
5      while G̃_active ≠ ∅ do
6          foreach G̃_i ∈ G̃_active do
7              stream ← DispatchStream(G̃_i)
8              if G̃_i is not resident in GPU memory then
9                  GBuf ← AllocateDeviceMemory( )
10                 TransferData(stream, GBuf, G̃_i, CPU2GPU)
11             Kernel (stream, G̃_i)
12         G̃_active ← FindActiveSubgraph(G)
   /* Graph Processing Kernel on the GPU */
13 Procedure Kernel(Subgraph G̃)
14     foreach v ∈ G̃.SetOfVertices do
15         if v is active then
16             foreach e ∈ v.outedges do
17                 if Update(v, e) = SUCCESS then
18                     Activate(e.destination_vertex)
```

Vertices reside in GPU memory
Edges are streamed to GPU on demand

A graph is partitioned into many slices

In an iteration, all active subgraphs will be transferred entirely to GPU and processed there.

A Generic Example of Graph Processing Engine

```
1  Procedure SimpleSubgraphEngine(Graph G)
2      Load G̃'s subgraphs in {G̃_1, ··· , G̃_n} into the host
3      VertexInitialization(G)
4      G̃_active ← FindActiveSubgraph(G)
5      while G̃_active ≠ ∅ do
6          foreach G̃_i ∈ G̃_active do
7              stream ← DispatchStream(G̃_i)
8              if G̃_i is not resident in GPU memory then
9                  GBuf ← AllocateDeviceMemory( )
10                 TransferData(stream, GBuf, G̃_i, CPU2GPU)
11             Kernel (stream, G̃_i)
12         G̃_active ← FindActiveSubgraph(G)

   /* Graph Processing Kernel on the GPU */
13 Procedure Kernel(Subgraph G̃)
14     foreach v ∈ G̃.SetOfVertices do
15         if v is active then
16             foreach e ∈ v.outedges do
17                 if Update(v, e) = SUCCESS then
18                     Activate(e.destination_vertex)
```

Vertices reside in GPU memory
Edges are streamed to GPU on demand

A graph is partitioned into many slices

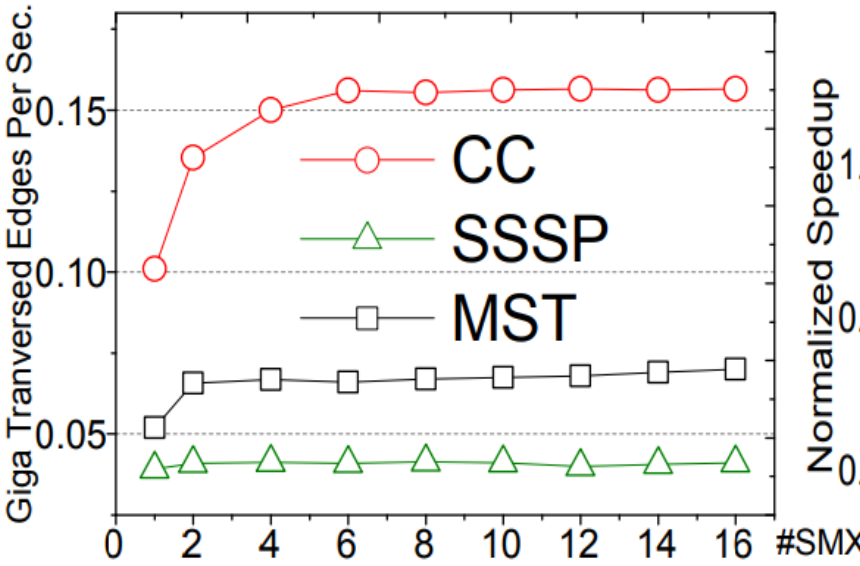
In an iteration, all active subgraphs will be transferred entirely to GPU and processed there.

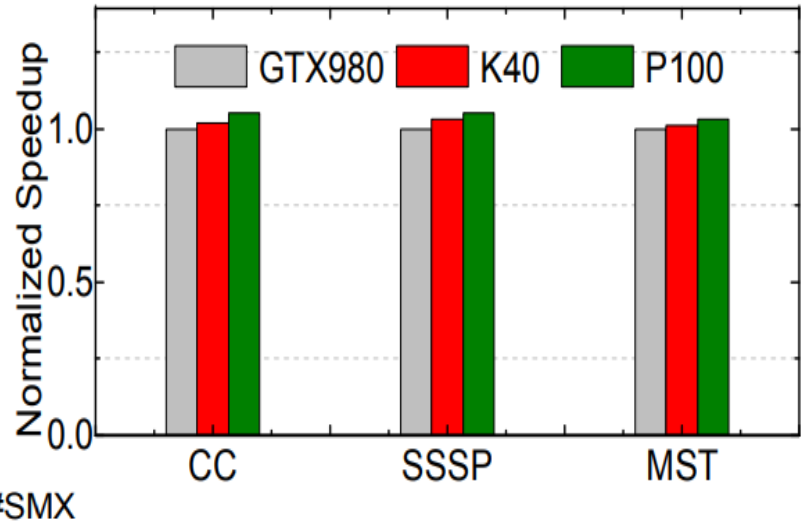
These active subgraphs processed on GPU will activate more destination vertices possibly.

Motivation

This simple graph engine wastes a considerable amount of limited host-GPU bandwidth, limiting performance and scalability further.

	Algo.	Used	Unused
TW	CC	12.15GB	21.44GB
	SSSP	22.74GB	77.42GB
	MST	25.78GB	106.27GB
UK	CC	43.41GB	688.43GB
	SSSP	81.64GB	1302.85GB
	MST	134.93GB	2099.25GB





Only 6.29%~36.17% transferred data are used

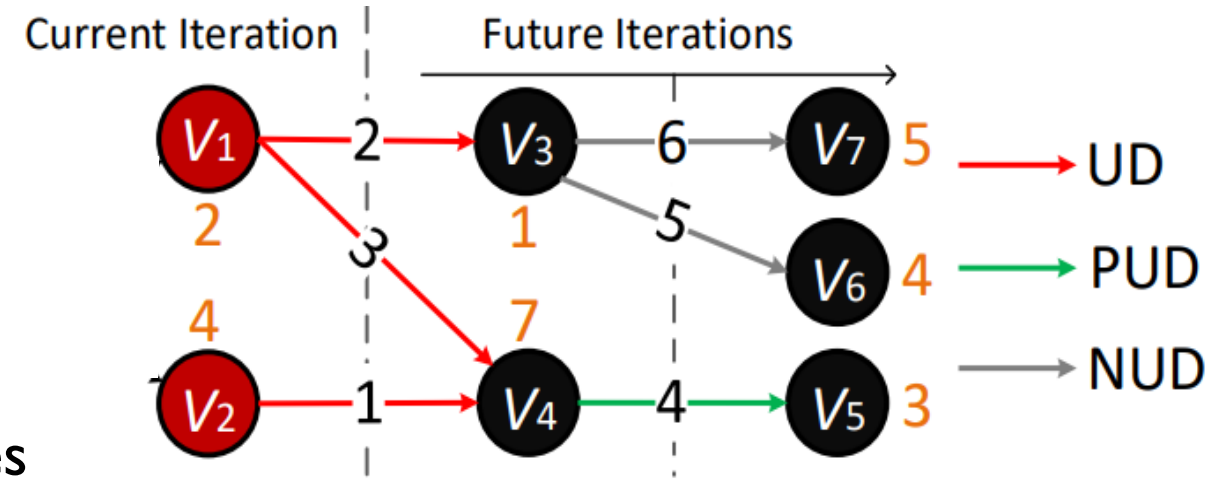
Perf. can be plateaued quickly at #SMX=4

Little gains when more powerful GPUs are used

Characterization of Subgraph Data

The data of a subgraph are changing

- Useful Data (UD)
  - associated with active vertices
  - must be transferred to GPU
- Potentially Useful Data (PUD)
  - associated with all future active vertices
  - (not) used in future (current) iteration
- Never Used Data (NUD)
  - Converged
  - Never be active again

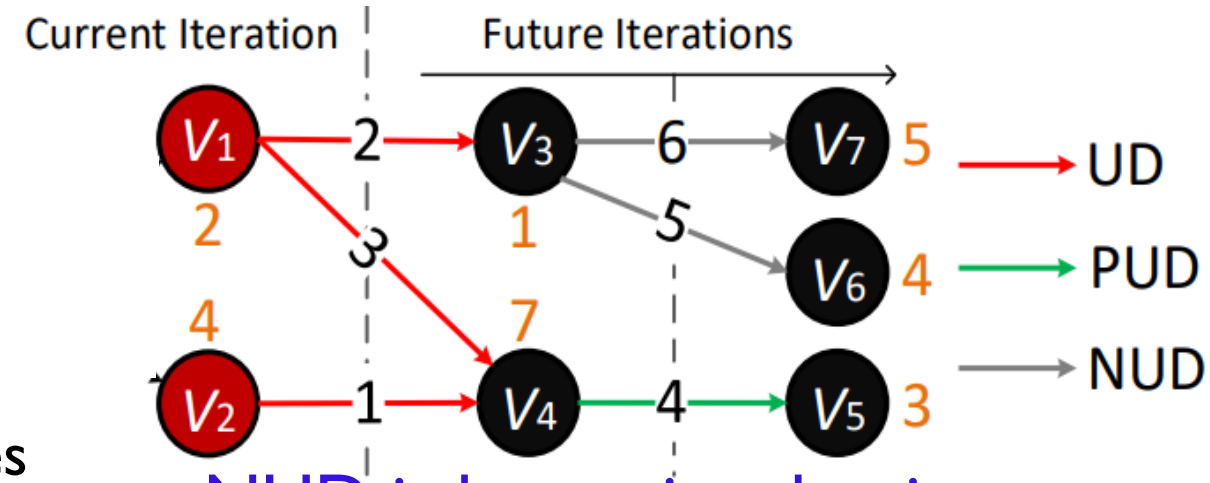


Characterization of Subgraph Data

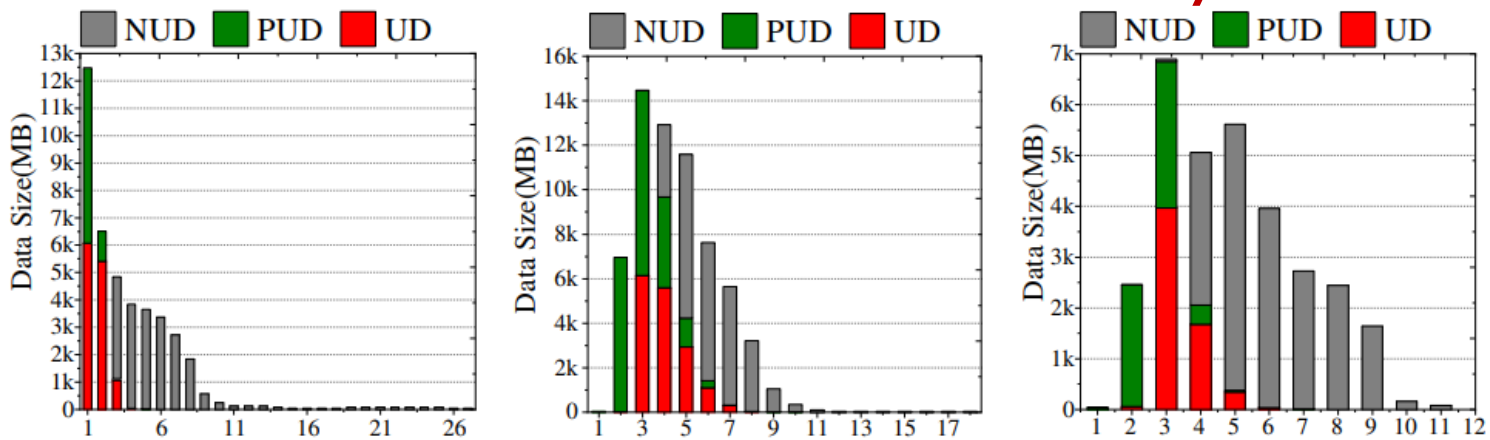
The data of a subgraph are changing

- Useful Data (UD)
  - associated with active vertices
  - must be transferred to GPU
- Potentially Useful Data (PUD)
  - associated with all future active vertices
  - (not) used in future (current) iteration
- Never Used Data (NUD)
  - Converged
  - Never be active again

PUD is substantial in earlier iterations but discarded



NUD is becoming dominant but streamed redundantly

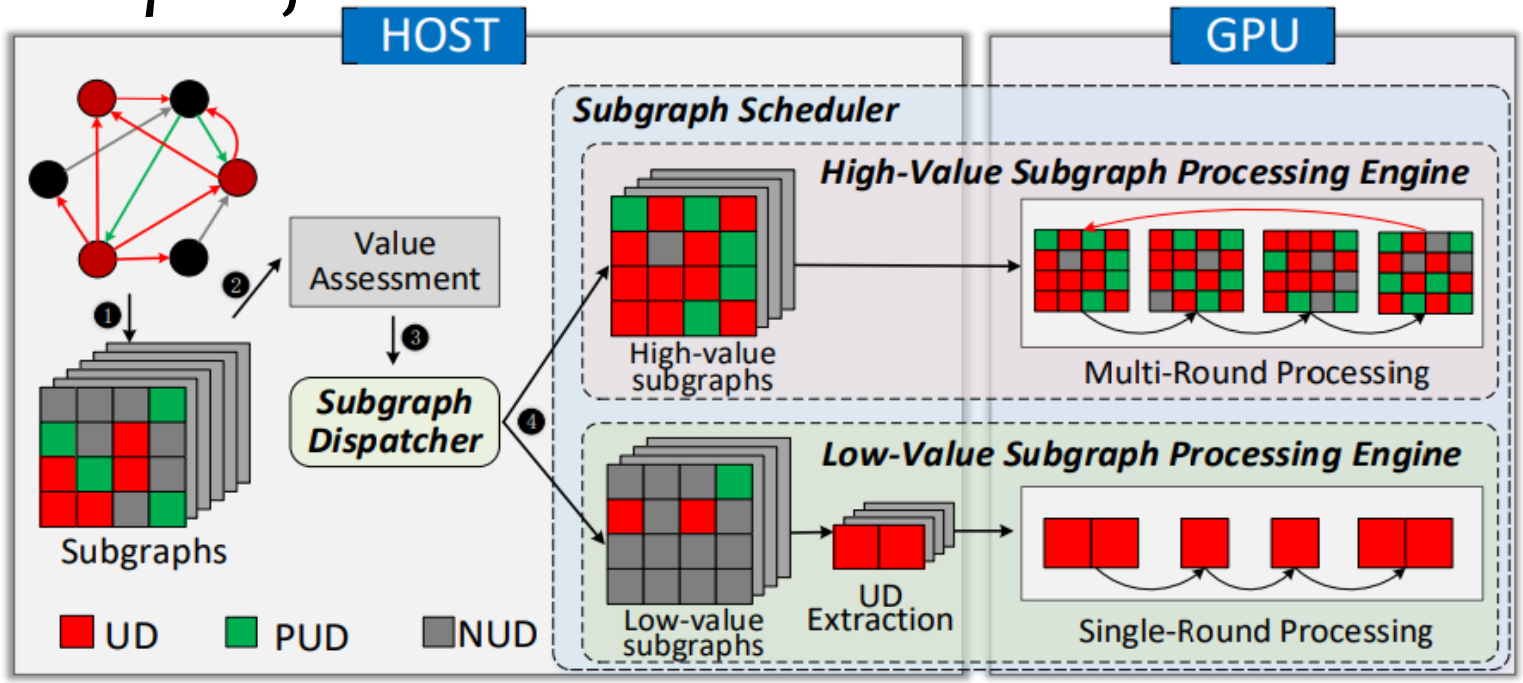


(a) CC (b) SSSP (c) MST

Contributions

Scaph *A scale-up graph processing for large-scale graph on GPU-accelerated heterogeneous platforms*

- Value-Driven differential Scheduling
  - distinguish high- and low-value subgraphs in each iteration adaptively



- Value-Driven Graph Processing Engines
  - exploit the most value out of high- and low-value subgraphs to maximize efficiency

Quantifying the Value of a Subgraph

- Conceptually, the value of a subgraph can be measured by its UD used in the current iteration and its PUD used in future iterations.
- The value of a subgraph from the current iteration and MAX-th iteration can be defined as:

$$Val(\tilde{G}) = \sum_{i=Cur}^{MAX} \sum_{v \in \tilde{G}.SetOfVertices} D(v) * A^{i}(v)$$

- The value of a subgraph depends upon its active vertices and their degrees

Value-Driven Differential Scheduling

```
1  Procedure VDDSEngine(Graph G)
2      Distribute G's subgraphs {G̃1,···,G̃n} to NUMA
         nodes
3      VertexInitialization(G)
4      G̃active ← FindActiveSubgraph(G)
5      Transfer VertexStates from GPU to CPU
6      while G̃active ≠ ∅ do
7          foreach G̃ ∈ G̃active do
8              if Predictor( G) = "HIGH-VALUE" then
9                  Push(HVworklist, G̃)
10             else
11                 Push(LVworklist, G̃)
12         HVSPEngine(HVworklist)
13         LVSPEngine(LVworklist, VertexStates)
14         G̃active ← FindActiveSubgraph(G)
15         Transfer VertexStates from GPU to CPU
```

- G is partitioned and distributed on NUMA nodes
- Vertices on GPU, edges streamed
- Estimate the value of an active subgraph
- Differential Scheduling
  - High-Value Subgraph Engine
  - Low-Value Subgraph Engine
- Updated vertices will be transferred from GPU to CPU. Edges, not modified, are not transferred

Checking If a Subgraph is High Value

- Suppose a subgraph *G* is a high-value subgraph, its throughput can be measured below:

$$T_{HV}(\tilde{G}) = \frac{|UD| + \lambda |PUD|}{|\tilde{G}|/BW + t_{barrier}}$$

- Suppose a subgraph *G* is a low-value subgraph, its throughput can be measured below:

$$T_{LV}(\tilde{G}) = \frac{|UD|}{|UD|/BW + t_{barrier}}$$

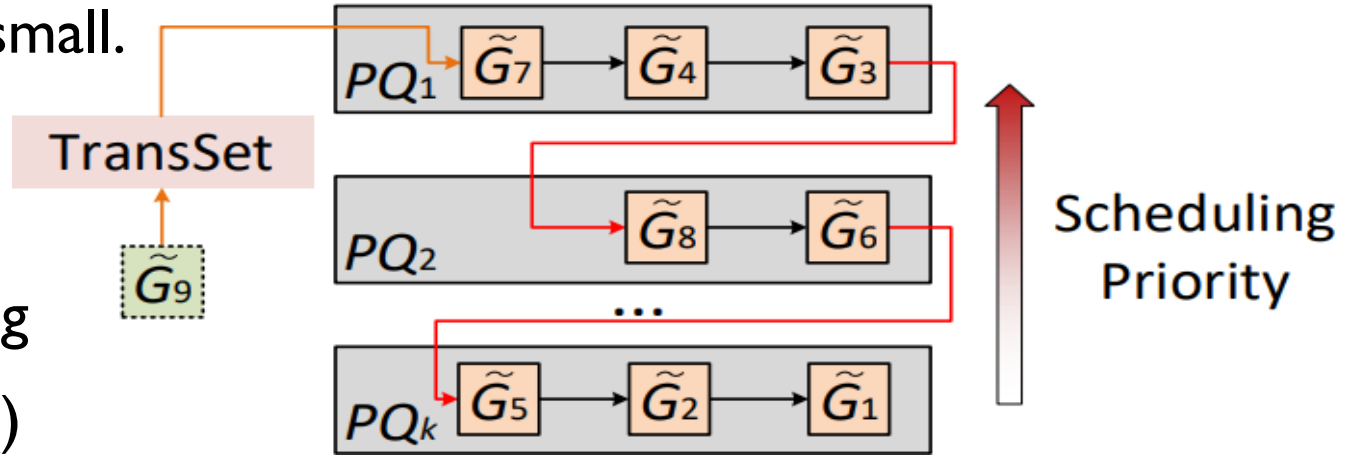
- Now, *G* is a high-value subgraph if $T_{HV}(\tilde{G}) \geqslant T_{LV}(\tilde{G})$. Thus, we need to analyze:

$$|UD| + \lambda |PUD|(1 + \frac{t_{barrier}}{|UD|/BW}) > |\tilde{G}|$$

- This condition is heuristically simplified below:
  - $|UD|/|\tilde{G})| > \alpha$, which indicates UD is dominant.
  - $|UD_{current}| - |UD_{last}| > 0$ and $|UD|/|\tilde{G}| > \beta$. UD remains a medium level but is growing increasingly over iteration.
  - α=50%, β=30%

High-Value Subgraph Processing

- Inspired from CLIP (ATC'17), each high-value subgraph can be scheduled multiple times to exploit *intrinsic* value of a subgraph
- In a GPU context, subgraph sizes are small.
- We propose a delayed scheduling to exploit PUD *across the subgraphs*
- Queue-assisted multi-round processing
  - *k*-level priority queue ($PQ_1, \ldots, PQ_k$)
  - Subgraph streamed to *TransSet* asynchronously
  - A subgraph in PQ_1 is scheduled first. Its priority drops by one once processed
  - Subgraph transfer and scheduling are executed concurrently



Complexity Analysis

- Time Complexity
  - The queue depth *k* is expected to be bounded by BW'/BW
  - For a typical server (BW'=224GB/s and BW=11.4GB/s), k can be less than 20, which is typically small.
- Space Complexity
  - k-level queue maintains only the indices of the active subgraphs
  - The worst complexity is $O\left(\frac{Mem_{GPU} \times \text{sizeof(SubgraphIndex)}}{|\tilde{G}|}\right)$
  - For P100 (GPU memory: 16GB, Index size: 4B, subgraph size: 32M), the space overhead of the queue is 2KB, which is small.

Low-Value Subgraph Processing

- NUMA-Aware Load Balancing
  - Intra-node load balancing: The UD extraction for each subgraph is done in its own thread.
  - Inter-node load balancing: A NUMA node is duplicated an equal number of randomly selected subgraphs from the other nodes
- Bitmap-based UD extraction
  - All vertices of a subgraph is stored in a bitmap
  - 1 (0) indicates the corresponding vertex is active (inactive)
- To reduce the fragmentation of the UD-induced subgraphs, we divide each chunk to store a subgraph into smaller tiles.

Limitations (More details in the paper)

- Graph Partition
  - A greedy vertex-cut partition
- Out-of-core solution
  - Using the disk as secondary storage is promising to support even larger graphs
- Performance Profitability

Experimental Setup

- Baselines
  - Totem, Graphie, Garaph
- Graph Size: 32MB
- Graph Applications
  - Typical algorithms: SSSP/CC/MST
  - Actual workloads: Two NNDR/GCS
- Datasets
  - 6 real-world graphs:
  - 5 large synthesized RMAT graphs
- Platforms
  - Host: E5-2680v4 (512GB memory, two NUMA nodes)
  - GPU: P100 (56 SMXs, 3584 cores, 16GB memory)

Dataset	#Vertices	#Edges	Avg. Degree	Size
twitter (TW)	41.7M	1.47B	39.5	32.8GB
comfriend (FR)	124.8M	1.81B	14.5	40.4GB
sk-2005 (SK)	50.6 M	1.95B	38.5	43.6GB
uk-2007 (UK)	105.9M	3.74B	35.3	83.6GB
altavista-2002 (AV)	1.41G	6.64B	4.695	148.3GB
fb-2009 (FB)	139.1M	12.3B	88.7	275.6GB
RMAT-k ($25<k<31$)	2^k	2^{k+4}	16	-

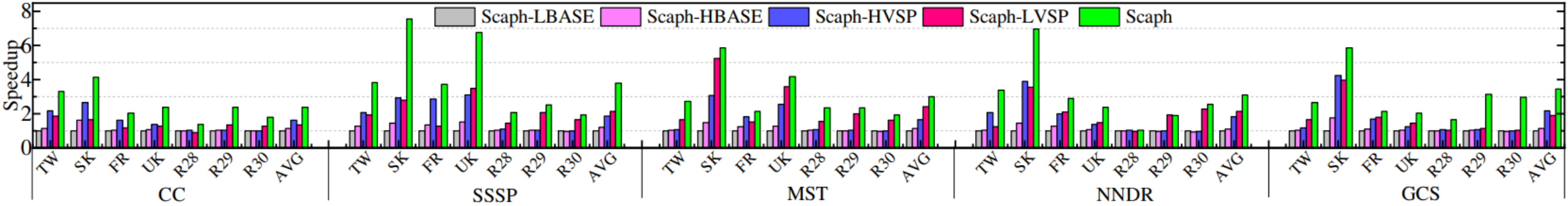
Efficiency

- Scaph vs. Totem
  - UD and PUD exploited more fully
  - yields 2.23x~7.64x speedups
- Scaph vs. Graphie
  - Exploit PUD and NUD is discarded
  - yields 3.03x~16.41x speedups
- Scaph vs. Garaph
  - Removing NUD transferred
  - yields 1.93x~5.62x speedups

Algorithm	System	Execution Time (Secs)					
		TW	FR	SK	UK	AV	FB
CC	Totem	2.41	5.01	2.72	9.32	N/A	N/A
	Graphie	1.89	4.46	16.53	23.61	57.49	133.21
	Garaph	1.17	2.53	2.90	7.07	31.46	86.24
	Scaph	0.28	0.91	1.08	2.47	7.08	15.35
SSSP	Totem	5.94	5.78	7.07	19.21	N/A	N/A
	Graphie	5.32	9.24	52.01	89.44	218.51	413.07
	Garaph	3.71	4.45	6.83	16.52	114.68	204.35
	Scaph	0.92	1.67	3.17	6.64	29.06	38.87
MST	Totem	7.93	10.90	21.33	42.84	N/A	N/A
	Graphie	8.45	16.24	32.19	53.22	198.85	304.51
	Garaph	4.14	7.38	12.35	25.82	101.25	131.45
	Scaph	1.39	1.99	2.93	6.36	25.23	35.41
NNDR	Totem	6.47	6.63	12.17	29.43	N/A	N/A
	Graphie	5.38	7.32	28.19	49.81	234.04	457.13
	Garaph	3.41	4.76	9.28	28.74	116.34	175.34
	Scaph	1.77	2.08	2.99	5.13	20.19	33.55
GCS	Totem	19.77	23.04	59.51	98.11	N/A	N/A
	Graphie	24.08	38.84	50.34	93.29	454.41	834.59
	Garaph	10.53	15.56	20.438	39.45	185.58	299.76
	Scaph	3.33	4.08	10.46	16.13	39.52	54.94

Effectiveness

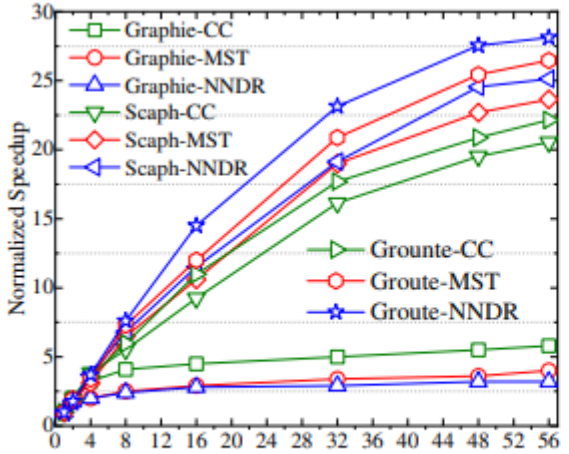
- Scaph-HVSP: All the low-value subgraphs are misidentified as high-value subgraphs
- Scaph-LVSP: All the high-value subgraphs are misidentified as low-value subgraphs
- Scaph-HBASE: Differential processing is used but queue-based scheduling is not applied
- Scaph-LBASE: A variation of Scaph-LVSP except that every subgraph is streamed entirely



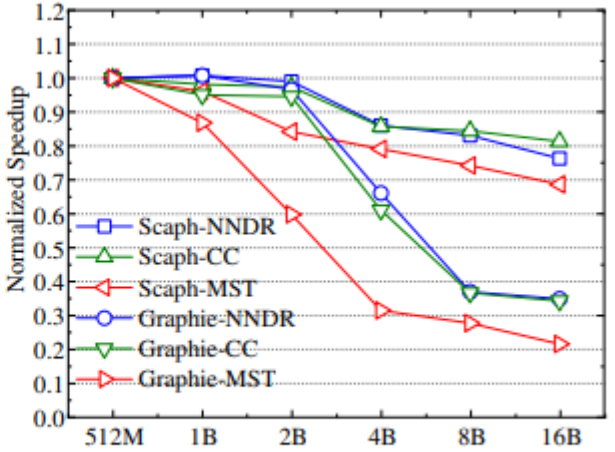
- Scaph-HBASE vs. Scaph-HVSP: Significant performance difference shows the effectiveness of our delay-based subgraph scheduling
- Scaph vs. Scaph-LVSP and Scaph-HVSP: Scaph obtains the best of both worlds, showing the effectiveness of differential subgraph scheduling

Sensitivity Study

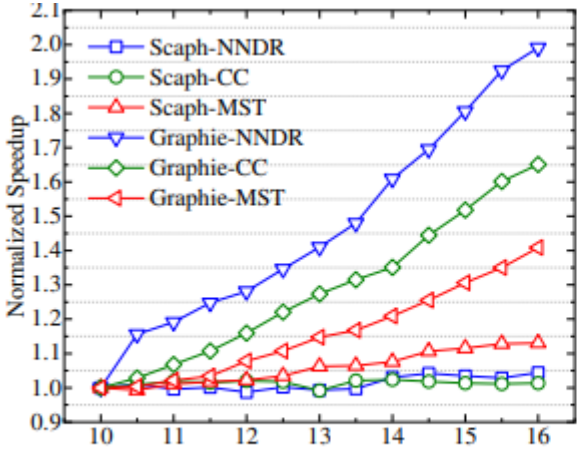
- Varying #SMXs
  - Significantly more scalable
- Varying Graph Sizes
  - Slower performance reduction rate
- Varying GPU memory
  - Scaph is nearly insensitive to the GPU memory used
- GPU generations
  - Enables significant speedups



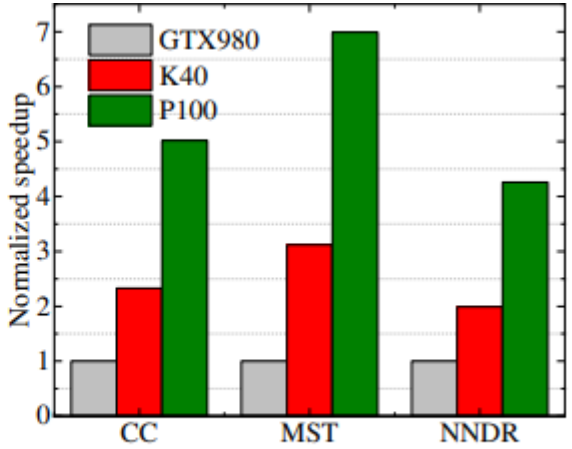
(a) #SMXs



(b) Edge sizes



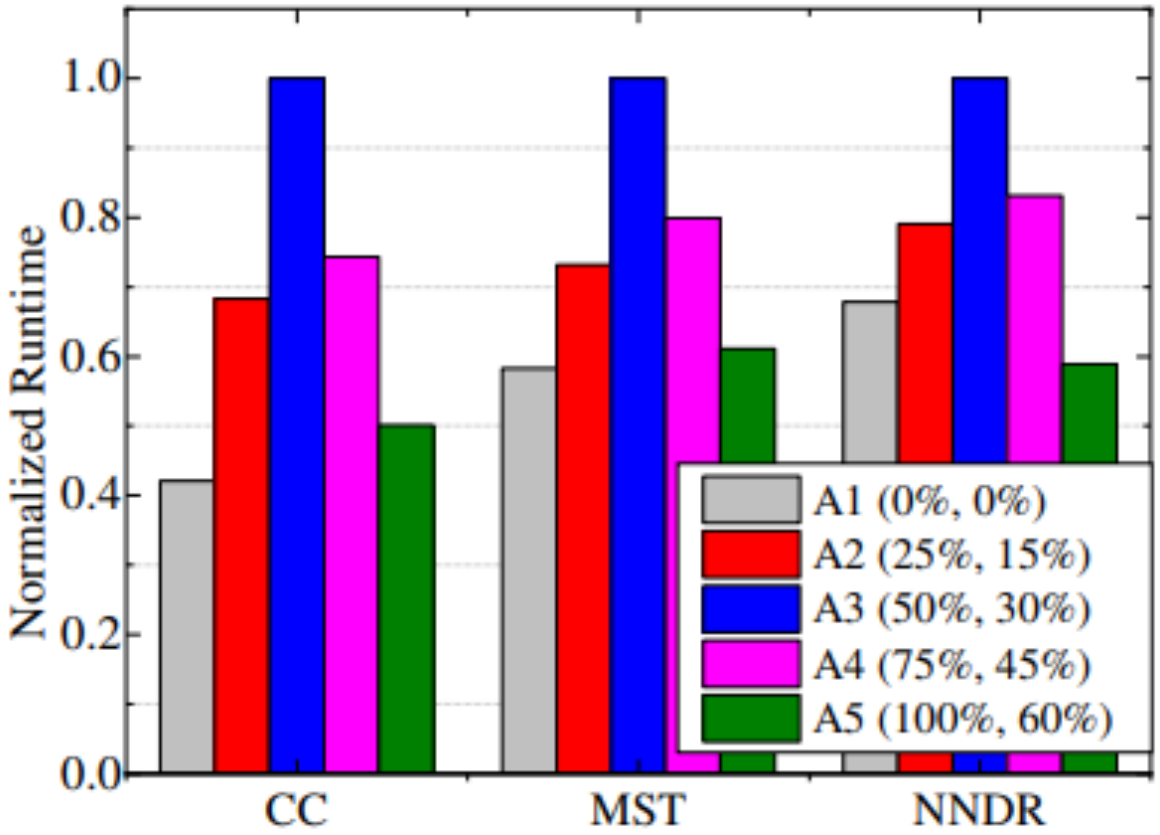
(c) GPU memory (GB)



(d) GPU generations

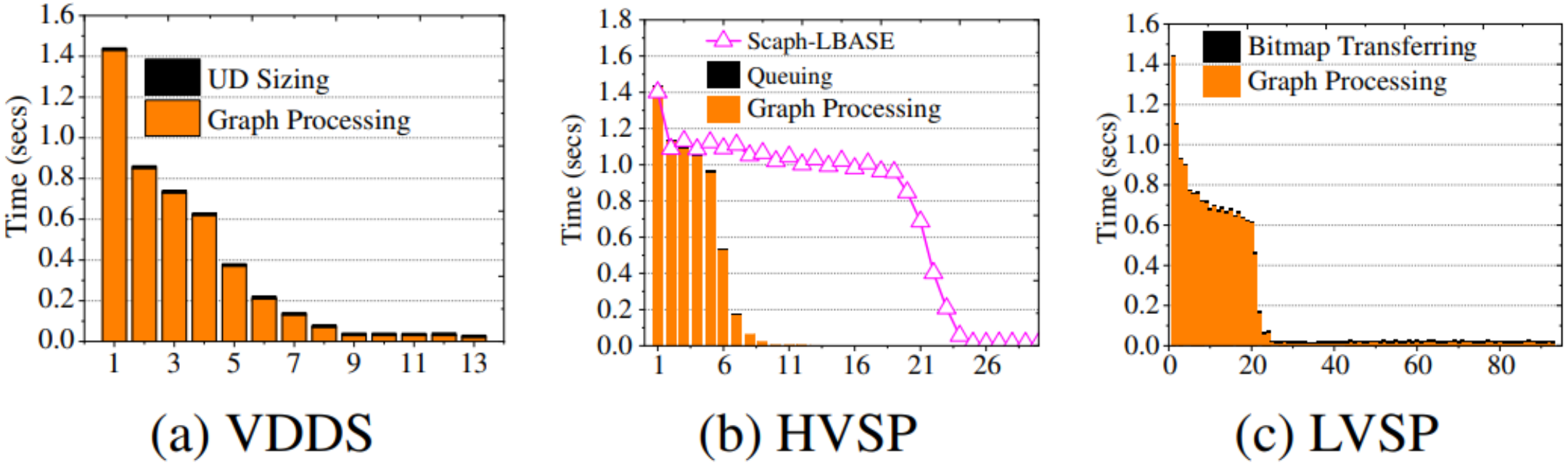
Sensitivity Study (con't)

- A1: Scaph-HVSP
- A5: Scaph-LVSP
- A3 represents a nice point for yielding good performance results.



(e) Varying α and β

Runtime Overhead



- **VDDS:** The cost of computing the subgraph value is negligible
- **HVSP:** Queue management cost per iteration is as small as 0.79% of total time
- **LVSP:** CPU-GPU bitmap transfer cost per iteration represents 4.3% of total time

Conclusion

Scaph: Scale up graph processing for large graphs on GPU-accelerated heterogenous architectures.

- Subgraph Value Characterization, which quantifies the value of a subgraph adaptively and dynamically
- Value-Driven Differential Scheduling, which adaptively distinguishes high- and low-value subgraphs and dispatches them to an appropriate graph processing engine
- Value-Driven Graph Processing Engines, which squeeze the most value out of high- and low-value subgraphs to maximize efficiency
- It outperforms state-of-the-art heterogeneous graph systems, Totem (4.12×), Graphie (8.93×), and Garaph (3.71×).

Thanks!

longzh@hust.edu.cn