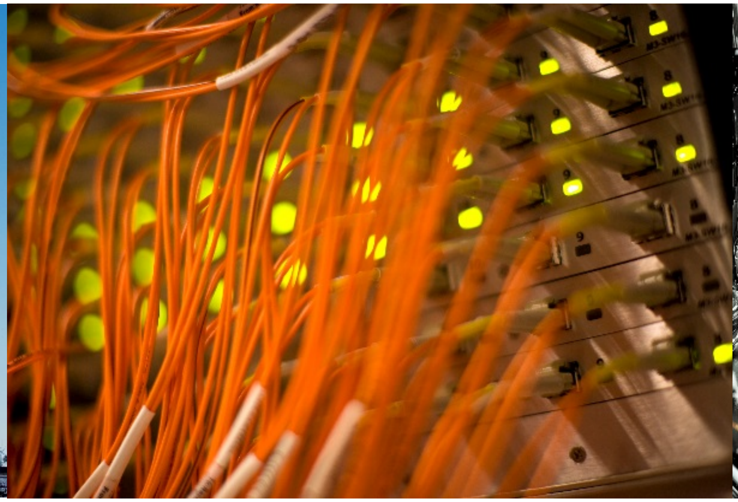
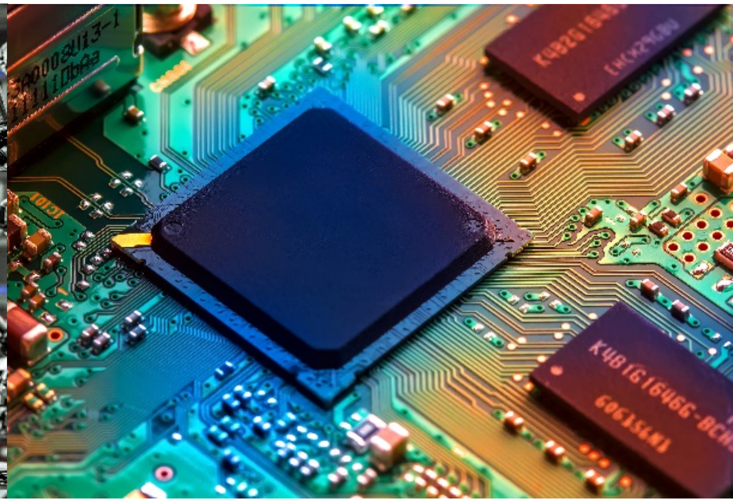
Update on K-truss Decomposition on GPU

Mohammad Almasri[1], Omer Anjum[1], Carl Pearson[1], Zaid Qureshi[2], Vikram S. Mailthody[1], Rakesh Nagi[3], Jinjun Xiong[4], Wen-mei Hwu[1]

[1] ECE, [2] CS, [3] ISE, University of Illinois at Urbana-Champaign, Urbana, IL 61801

[4] Cognitive Computing & University Partnership, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598





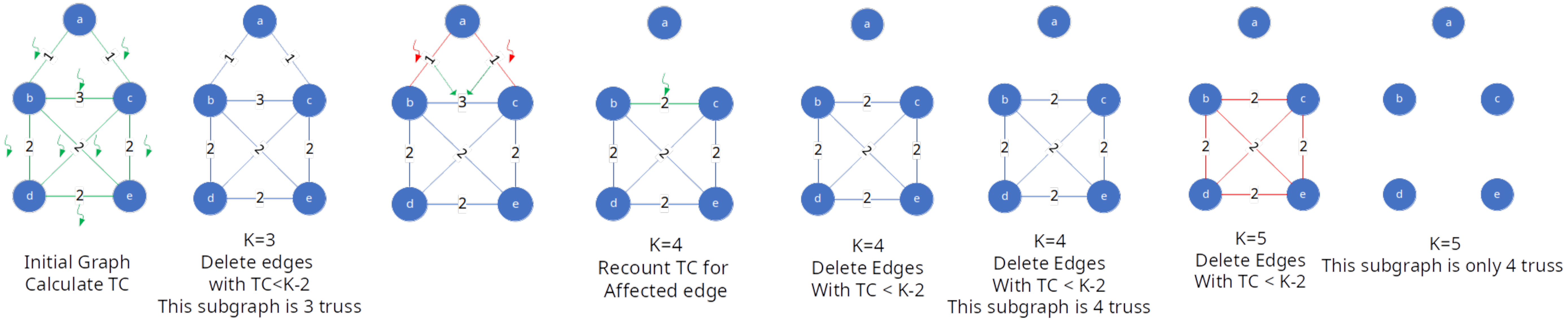
ILLINOIS

Electrical & Computer Engineering

COLLEGE OF ENGINEERING

Introduction

- **k-Truss**: is a cohesive subgraph in which each edge is part of at least k−2 triangles [1,2].
- This subgraph relaxes the concept of clique and can be computed in polynomial time.
- k-truss decomposition, peeling approach:



Single-GPU Optimizations (1/2)

Our 2018 implementation[3]:

```
k = kmin
while(true)
   while(num_affected_edges > 0)
      ktruss_kernel(k, edges, deleted, affected, …)
      num_affected_edges = count(affected)
   num_deleted_edges = count(deleted)
   if(num_deleted_edges == num_edges)
      break
   else
      edges = stream_compaction(deleted, edges)
      num_edges = num_edges - num_deleted_edges
      k = k + 1
```

Unnecessary step !

Expensive operation !

Our 2019 implementation:

```
k = kmin
while(true)
   while(any_affected)
      any_affected = ktruss_kernel(k, edges, deleted, …)

   num_deleted_edges = reduce_add(deleted)
   if(num_deleted_edges == num_edges)
      break
   else
      if(num_deleted_edges/num_edges  > threshold)
       edges = stream_compaction(deleted, edges)
       num_edges = num_edges - num_deleted_edges
      k = k + 1
```

'deleted' list: holds a flag for each edge to indicate whether the edge is deleted.
'affected' list: holds a flag for each edge to indicate whether the edge is affected by the deletion of any other edge with which it shares triangles.

Single-GPU Optimizations (2/2)

2018 implementation:

```
function ktruss_kernel(k, edges, deleted, affected, …)
   foreach (e in edges)
      if(!deleted[e] && affected[e])
         tc = triangle_count(e)
         if(tc < k-2)
            deleted[e] = true
            affect_edges(e)
      ….
```

```
function triangle_count(e)
   u = get_left_node(e)
   v = get_right_node(e)
   intersections = intersect(adj(u), adj(v))
   return count(intersections)
```

```
function affect_edges(e)
   u = get_left_node(e)
   v = get_right_node(e)
   intersections = intersect(adj(u), adj(v))
   foreach(i in intersections)
      affected[i] = true
```

Both *triangle_count* and *affect_edges* perform the same list intersection.

In 2019 implementation:

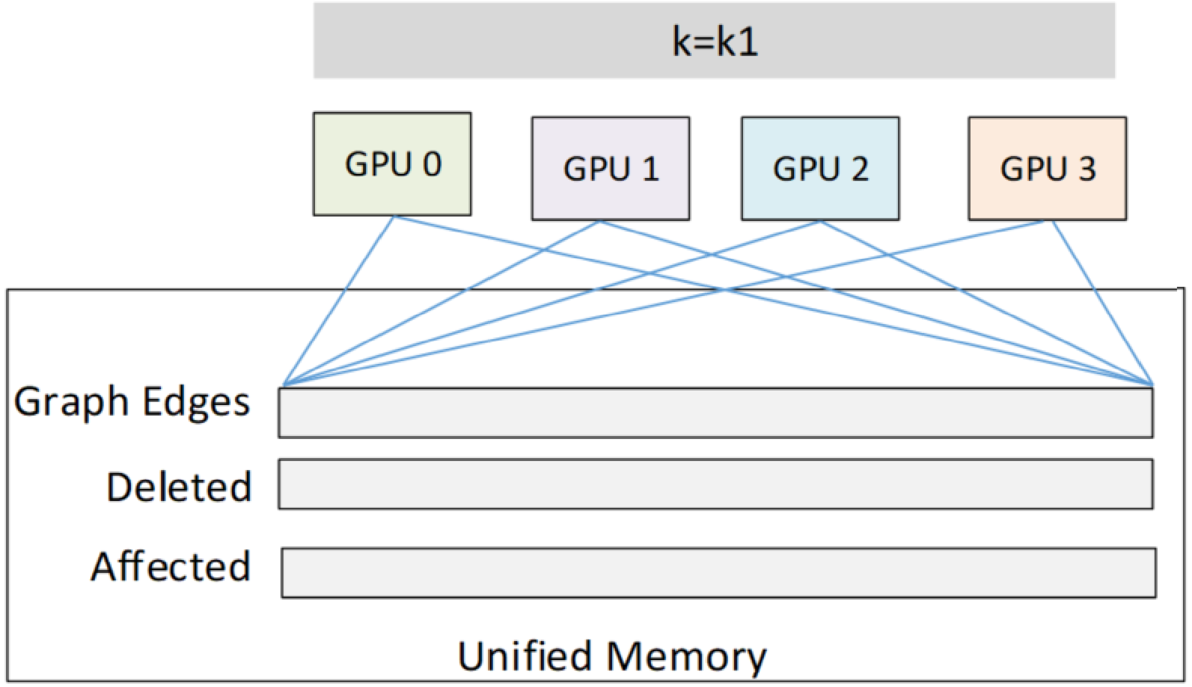
a) While doing triangle counting, record the indices of first and last intersections of the two adj. lists and use them in 'affect_edges' step:

```
function affect_edges(e, u_first, u_last, v_first, v_last)
      intersections = intersect(adj(u), adj(v), u_first,…)
```

b) In the triangle counting step, we start marking edges as 'affected' early once there is no hope to find k-2 triangles.

Multi-GPU Implementation

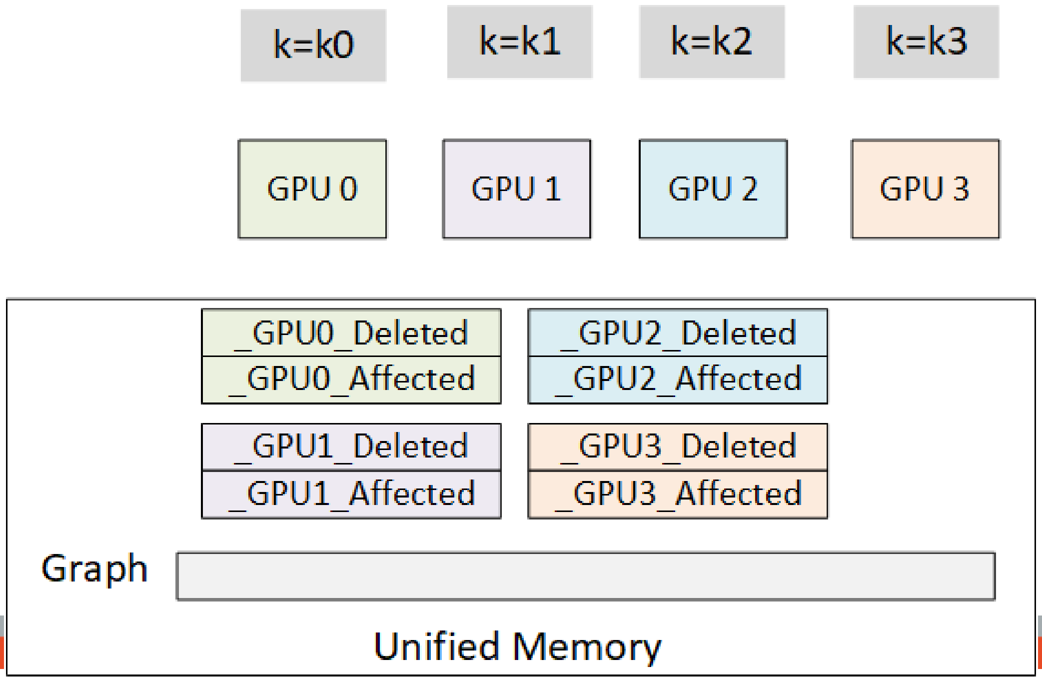
In 2018 submission:



Due to list intersection operations:
Graph, 'Deleted', and 'Affected' lists are accessed randomly by all GPUs → many redundant data transfers → significant slowdown as we scale GPUs.

In 2019 submission:

- During the ktruss_kernel: graph data is read-only. 'cudaMemAdviseReadMostly' prevents redundant transfers of read-only data.
  - **Slow migrations reduced and performance greatly improved.**
- 'Deleted' and 'Affected' lists are read/write.
- Parallelize across k values:



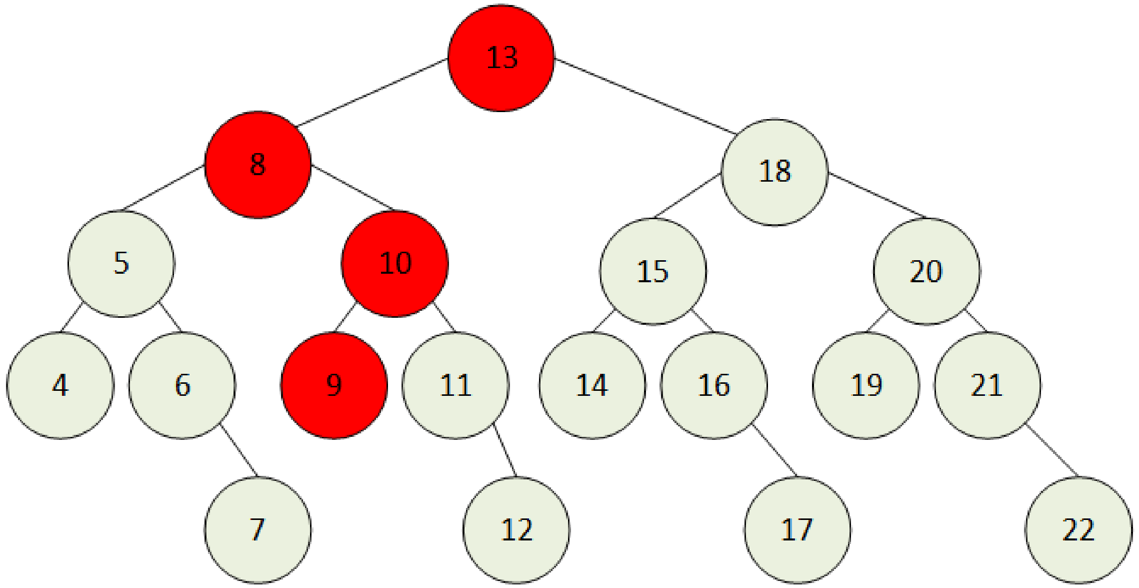
Binary-Search Approach to Find Maximum k

Algorithm:

- Evaluate for $k=(k_{upper\_bound} + k_{min})/2$.
- If the graph is not empty, do stream compaction and set $k_{min} = k$
- Else, revert to previous state and set $k_{upper\_bound} = k$
- Stop when $k_{upper\_bound} - k_{min} <= 1$.

Example: $k_{upper\_bound} = 23$, $k_{min} = 3$

If $k_{max}=9$



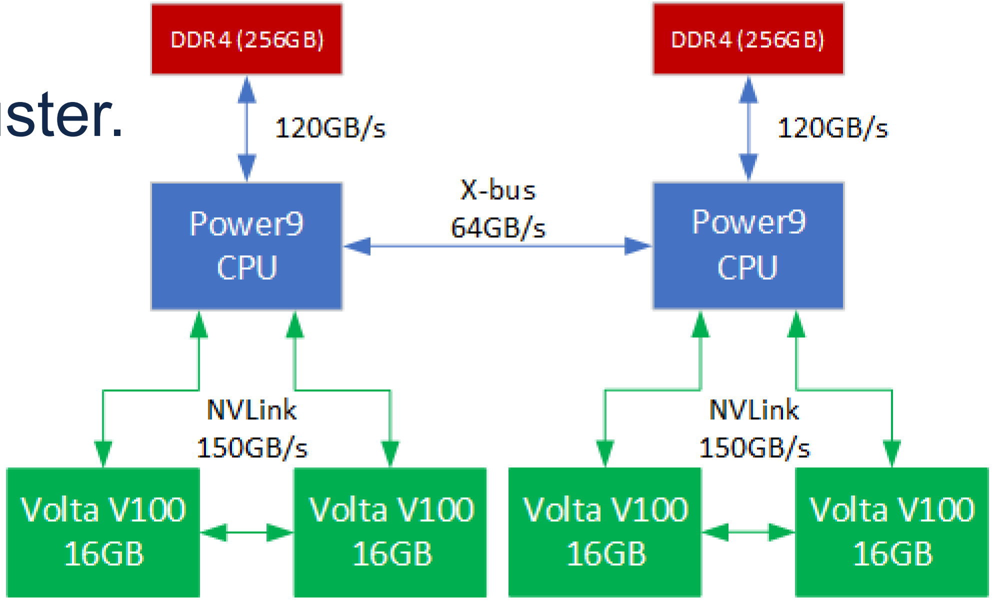
How to estimate k_{upper_bound}? Find the largest degree d for which there are at least d + 1 nodes, $k_{upper\_bound} = d+1$.

Additional optimizations:

- Two evaluated k values can be far apart → before evaluating k, eliminate nodes with degree < k.
- To process large graphs, such as Twitter with 2.8B bidirectional edges:
  - Empirically, $k_{max} > 5\%$ of k_{upper_bound}. Thus, before the first iteration, we remove nodes with out-degree < 5% of k_{upper_bound}.

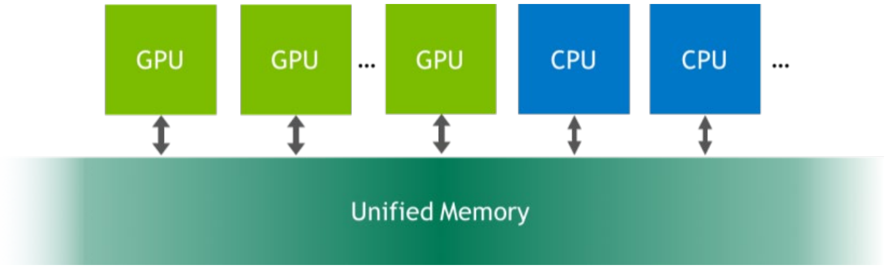
Evaluation Platform

- A node with Newell architecture from the NCSA HAL cluster.
  - 2 IBM Power9 CPUs each with
    - 20 Cores
    - 256GB of Memory
  - 4 NVIDIA Tesla V100 GPUs
  - CPUs & GPUs connected via NVLINK 2.0

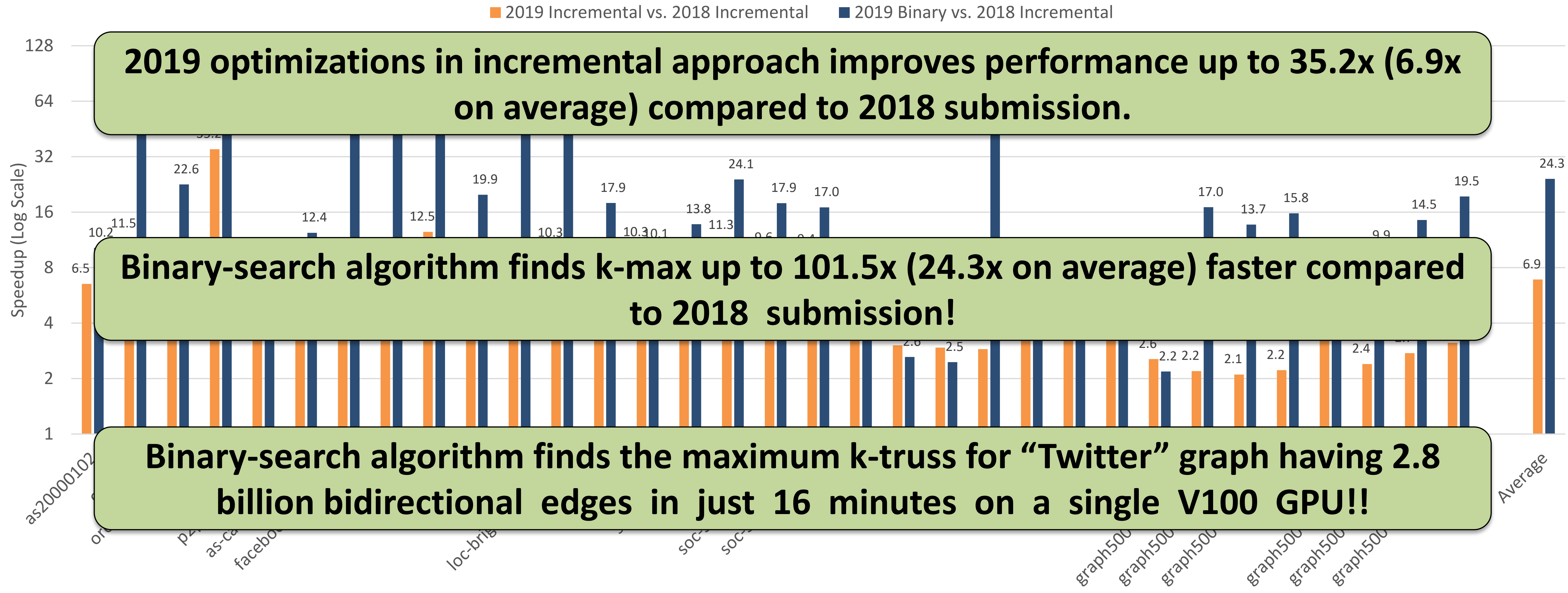


Memory Management:

- All auxiliary data structures are stored in the unified memory.
- Allocated using *cudaMallocManaged*.
- CUDA unified memory hints:
  - *cudaMemAdviseSetReadMostly*
  - *cudaMemAdviseUnsetReadMostly*



Single-GPU results

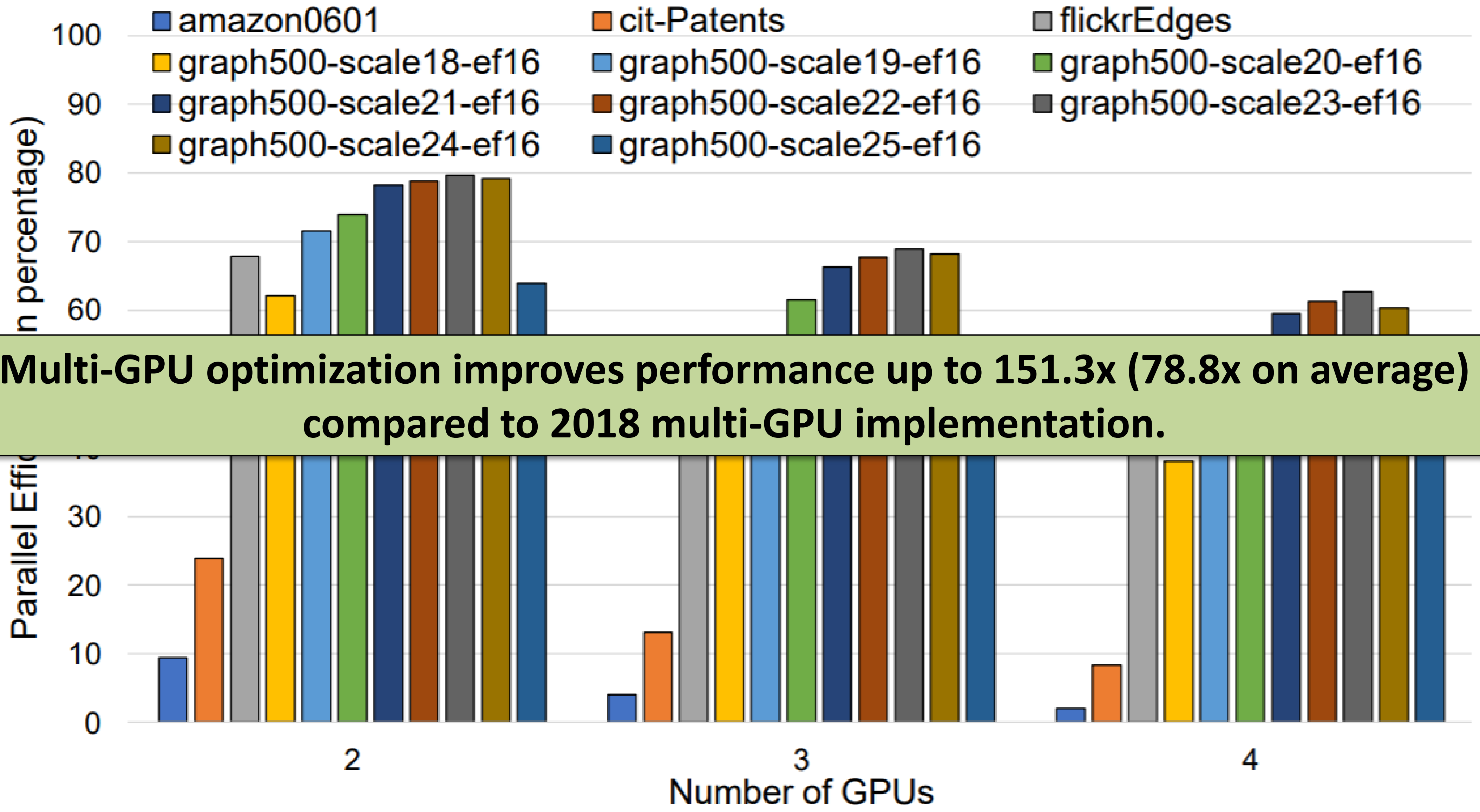


2019 optimizations in incremental approach improves performance up to 35.2x (6.9x on average) compared to 2018 submission.

Binary-search algorithm finds k-max up to 101.5x (24.3x on average) faster compared to 2018 submission!

Binary-search algorithm finds the maximum k-truss for "Twitter" graph having 2.8 billion bidirectional edges in just 16 minutes on a single V100 GPU!!

Multi-GPU Parallel Efficiency



Multi-GPU optimization improves performance up to 151.3x (78.8x on average) compared to 2018 multi-GPU implementation.

Conclusion

- Optimizations for the single-GPU implementation: limiting unnecessary compactions, reductions, and list intersection comparisons.
- Scalable multi-GPU implementation by using memory hints and parallelizing across k.
- Maximum k-truss, through binary-search rather than the incremental approach.

Compared to our 2018 work [3]:

Single-GPU:

Our incremental approach improves performance up to **35.2x** (**6.9x** on average).

Our binary approach improves performance up to **101.5x** (**24.3x** on average).

Multi-GPU:

We improve performance up to **151.3x (78.8x** on average).

The binary-search finds k_{max} for “Twitter” graph (2.8B bidirectional edges) in just 16 minutes on a single V100 GPU.

References

[1] J. Cohen. *Trusses: Cohesive subgraphs for social network analysis*. In *National Security Agency Technical Report*, page 16, 2008.

[2] J. Cohen. *Graph twiddling in a MapReduce world.* In *Computing in Science & Engineering*, 11(4):29-41, 2009

[3] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W. Hwu, “Collaborative (cpu + gpu) algorithms for triangle counting and truss decomposition,” in 2018 IEEE High Performance extreme Computing Conference (HPEC), Sep. 2018, pp. 1–7.

Thanks ....