A Dataflow-Graph Partitioning Method for Training Large Deep Learning Models

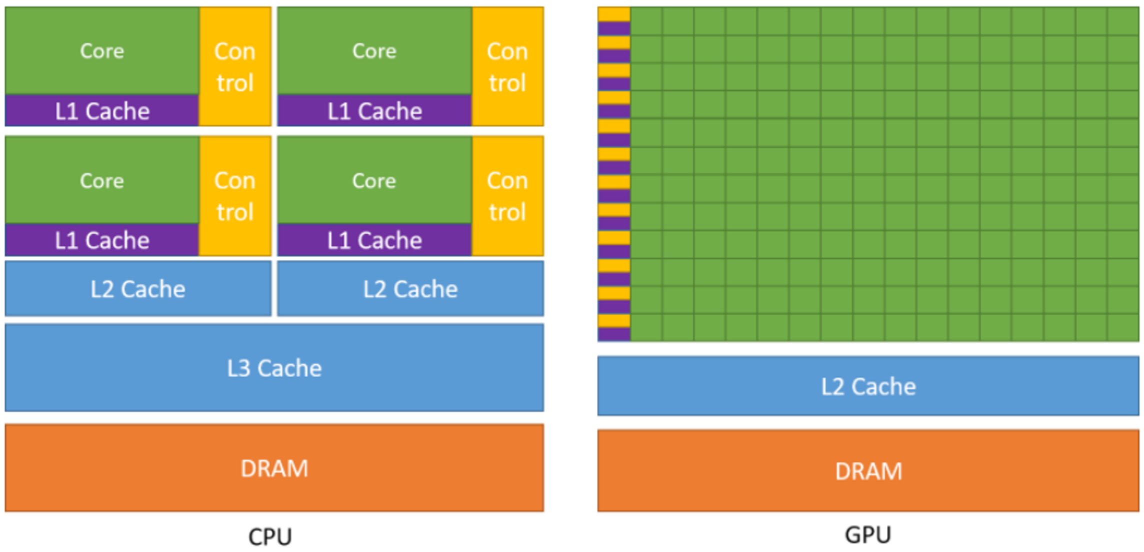
Didem Unat

Koç University, Istanbul, Turkey

ROSS Workshop 13/11/2020

DL Needs Throughput-Oriented Architecture

- DL models are compute intensive
- GPUs played major role in the renaissance of DL
  - Order of magnitude faster training
  - Many cores
  - High bandwidth memory



Memory Bottleneck

- Accelerators (GPUs) have a limited device memory
  - GPU V100 comes with 32 GBs
  - Technology limitations and price

Memory Bottleneck

- Accelerators (GPUs) have a limited device memory
  - GPU V100 comes with 32 GBs
  - Technology limitations and price

- DNNs grow in size
  - Higher accuracy on more complex tasks (Transformers)
  - Faster training
    - Wide ResNet vs ResNet
    - WRN-16-8 >> ResNet-101

Memory Bottleneck

- Accelerators (GPUs) have a limited device memory
  - GPU V100 comes with 32 GBs
  - Technology limitations and price

- DNNs grow in size
  - Higher accuracy on more complex tasks (Transformers)
  - Faster training
    - Wide ResNet vs ResNet
    - WRN-16-8 >> ResNet-101

- Models barely fit into single GPU memory
  - Use small batch sizes
    - Resource underutilization
- Models do not fit into single GPU memory

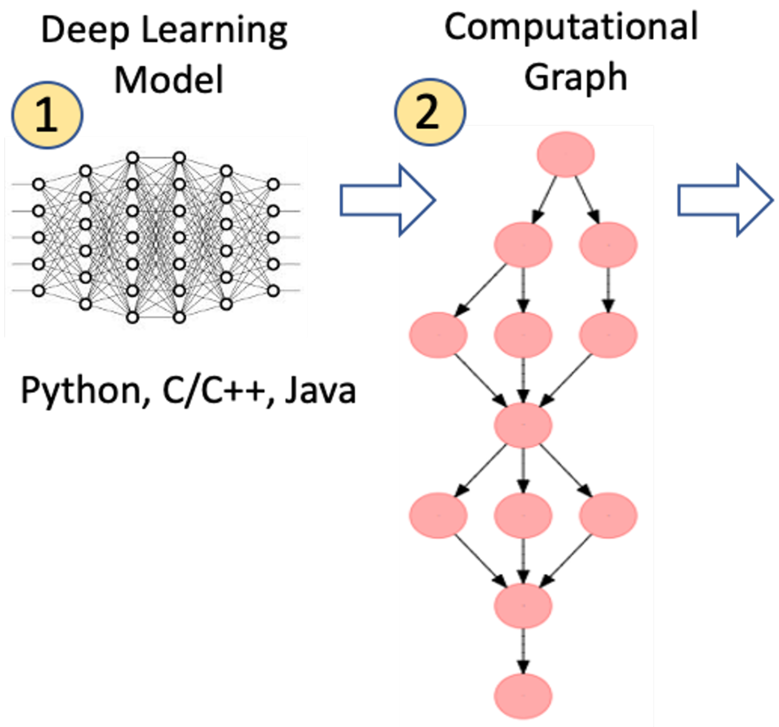
Related Work

- (1) Single device based solutions
  - Memory optimization techniques (Gradient Checkpointing)
  - Utilizing the host memory (Unified Memory)
- (2) Distributed training
  - Data parallelism
    - Doesn’t address the memory issue
  - Model parallelism (Gpipe, Pipedream, and others)
    - Model-specific, not general
    - Accuracy issues, requires manual tuning/implementations
  - Hybrid parallelism (Mesh-TensorFlow)
    - Specific, requires manual tuning

Our Approach: ParDNN

- Generic
  - Zero dependency and requires no knowledge about the DL aspects of the DNN models
- Automated, non-intrusive
  - Requires no modification of the model or operation kernels
- Works at system-level
  - Operates on computational graph

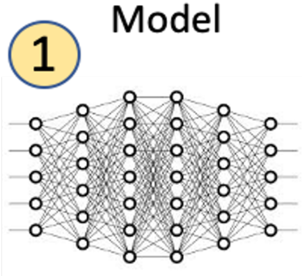
Computational Graph



- Operations in the graph represent one step
  - Both forward pass and back propagation are in the graph
- The graph is **static**
  - Constructed before running and stays the same
  - There are dynamic cases
- The graph is acyclic

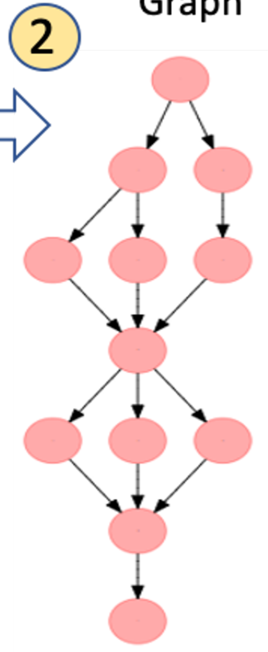
Computational Graph

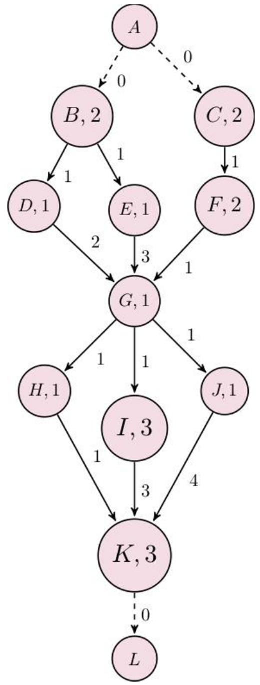
Deep Learning Model



Python, C/C++, Java

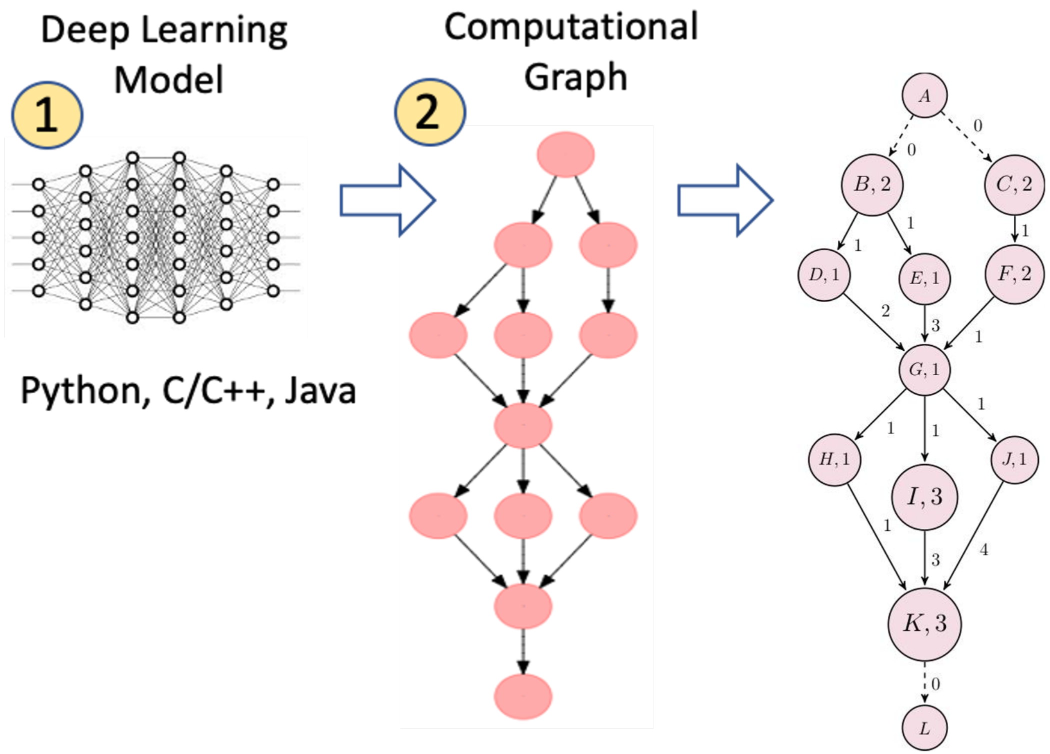
Computational Graph





- *G(V,E)*: Task graph
- *V*
  - *n* $\in$ *V*: Task.
  - *w(n)*: weight of n, computation time
- *E*
  - *e* $\in$ *E*: Dependency.
  - *c(e):* cost of e, communication time
  - Defines the execution order

Computational Graph



- *G(V,E)*: Task graph
- *V*
  - *n ∈ V*: Task.
  - *w(n)*: weight of n, computation time
- *E*
  - *e ∈ E*: Dependency.
  - *c(e):* cost of e, communication time
  - Defines the execution order

How to partition this task graph among multiple GPUs?

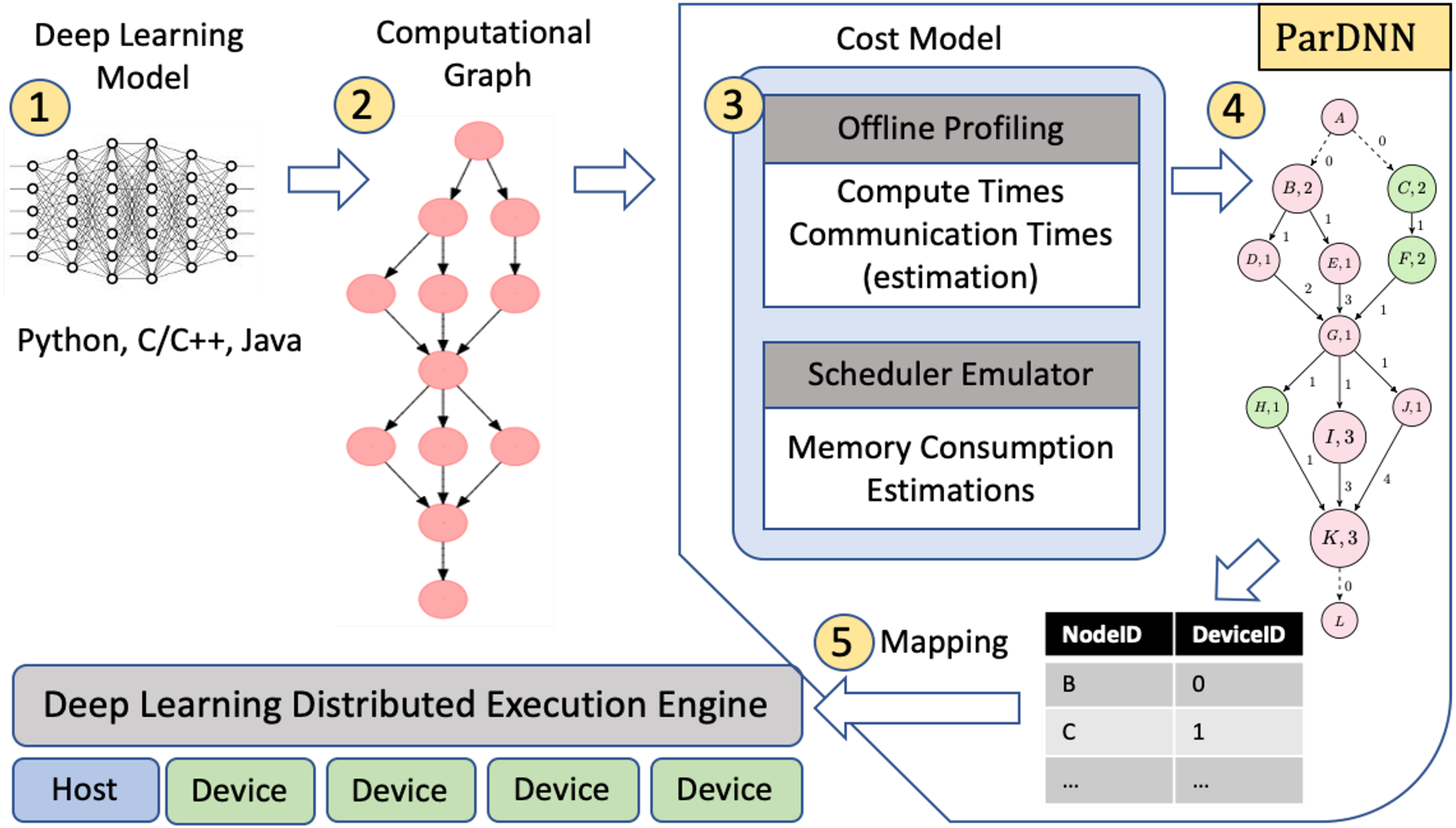
- *obey the memory constraints,*
- *reduce communication,*
- *minimize execution time*

Real DNN Graphs

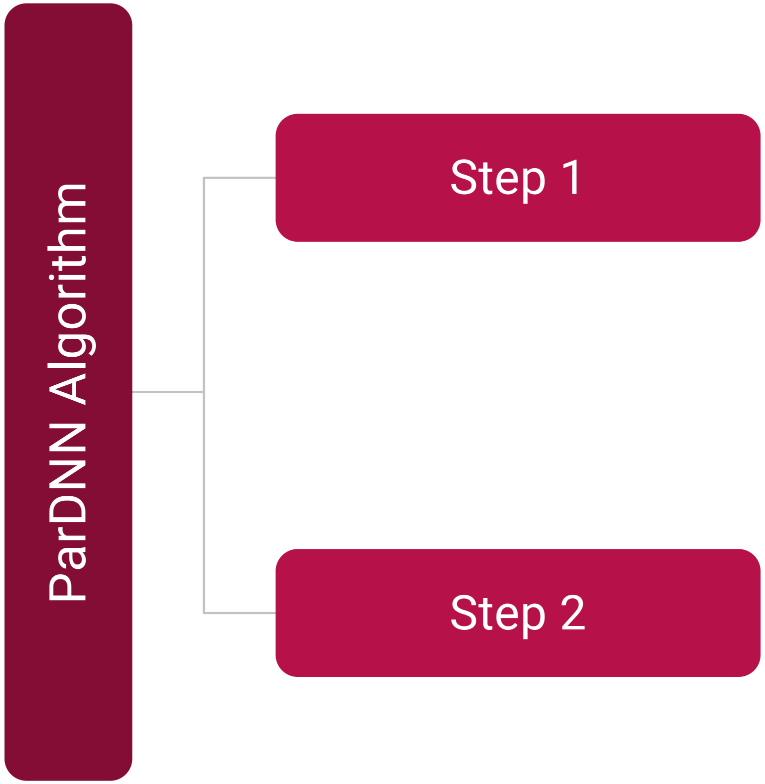
- Number of operations reaches hundreds of thousands, may scale up to millions.
  - **Another objective**: Low complexity is necessary

Model	Acronym	#Layers	HSD	SL	#Parameters	#Graph Nodes	Dataset
Recurrent Neural Network for Word-Level Language [51]	Word-RNN	10	2048	28	0.44 billion	11744	Tiny Shakespeare [23]
	Word-RNN-2	8	4096	25	1.28 billion	10578	
		#Layers	CHSD	ED			
Character-Aware Neural Language Models [26]	Char-CRN	8	2048	15	0.23 billion	22748	Penn Treebank (PTB) [33]
	Char-CRN-2	32	2048	15	1.09 billion	86663	
		#Conv. Layers [65]	#RU	WF			
Wide Residual Network [64]	WRN	610	101	14	1.91 billion	187742	CIFAR100 [28]
	WRN-2	304	50	28	3.77 billion	79742	
		#Layers	HSD	MD			
Transformer [54]	TRN	24	5120	2048	1.97 billion	80550	IWSLT 2016 German–English corpus [6]
	TRN-2	48	8192	2048	5.1 billion	160518	
		#Hidden Layers	FS				
Eidetic 3D LSTM[58]	E3D	320	5		0.95 billion	55756	Moving MNIST digits [50]
	E3D-2	512	5		2.4 billion	55756	

Our Approach: ParDNN

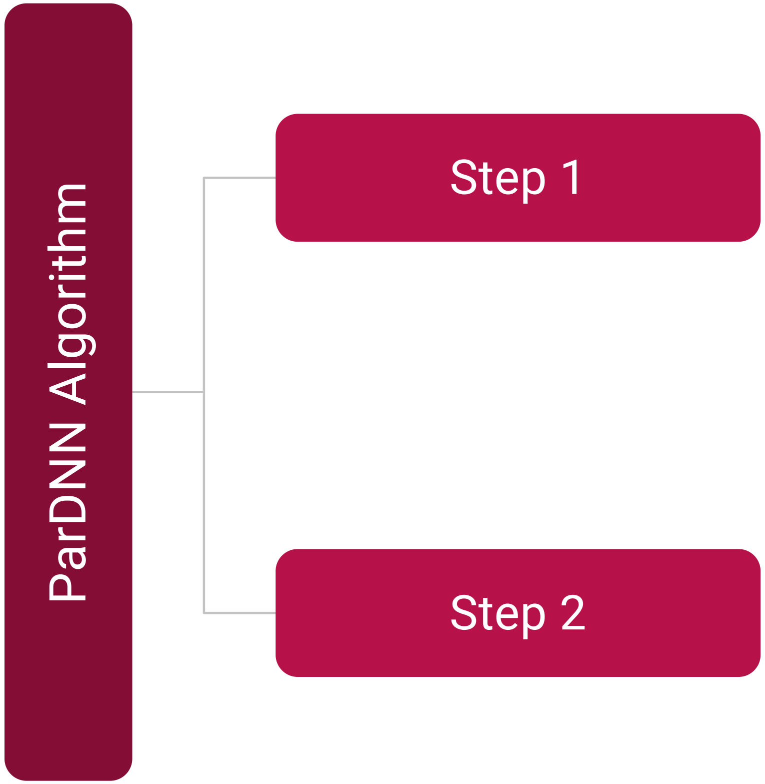


ParDNN Algorithm Overview



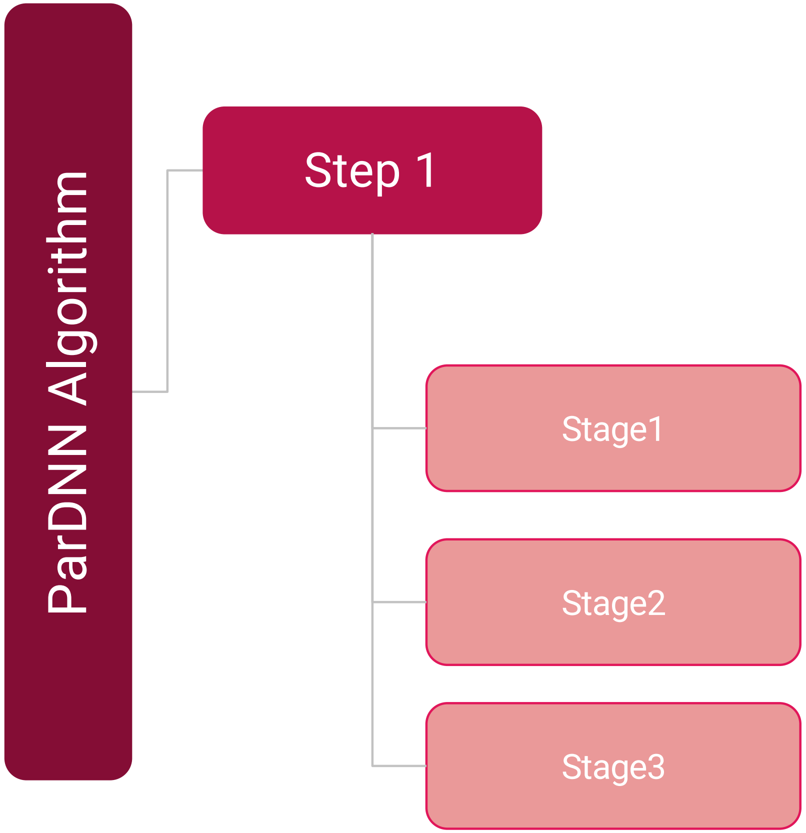
- Step 1: Given *K* devices, partition the graph into *K* partitions so that execution time is minimized
    - Communication time is minimized
    - Computation loads are balanced

ParDNN Algorithm Overview



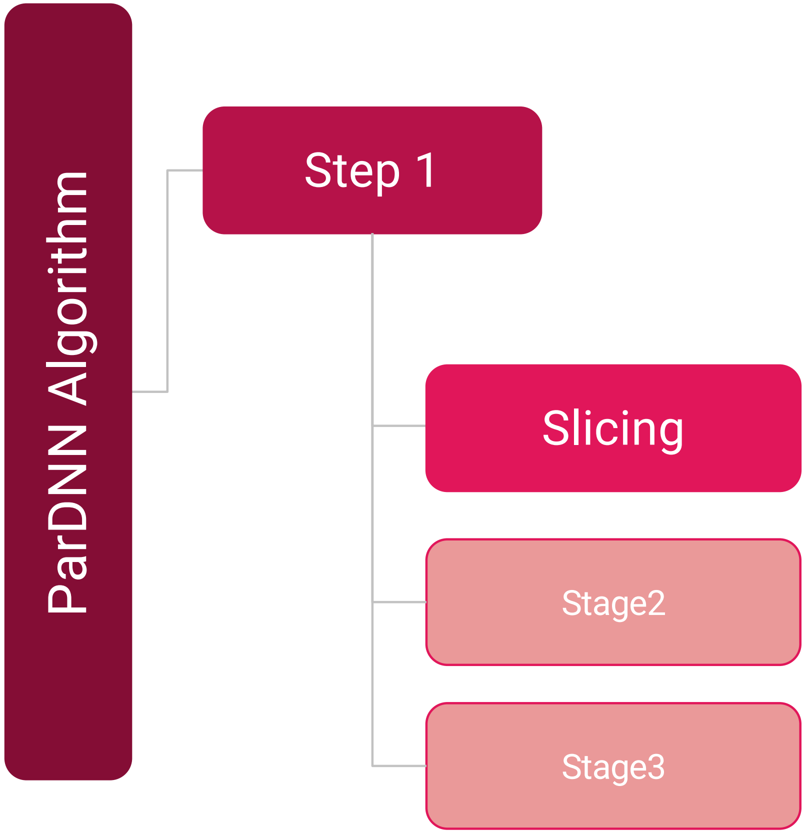
- Step 1: Given *K* devices, partition the graph into *K* partitions so that execution time is minimized
  - Communication time is minimized
  - Computation loads are balanced
- Step 2: Meet the memory consumption constraints
  - If each partition meets the device memory constraints
    - Done.
  - Else
    - Handle the memory overflow while maintaining locality-parallelism trade-off.

ParDNN Algorithm



- To achieve both
  - Good quality partitions
  - Reasonable runtime
- Step 1 is divided into 3 stages

ParDNN Algorithm

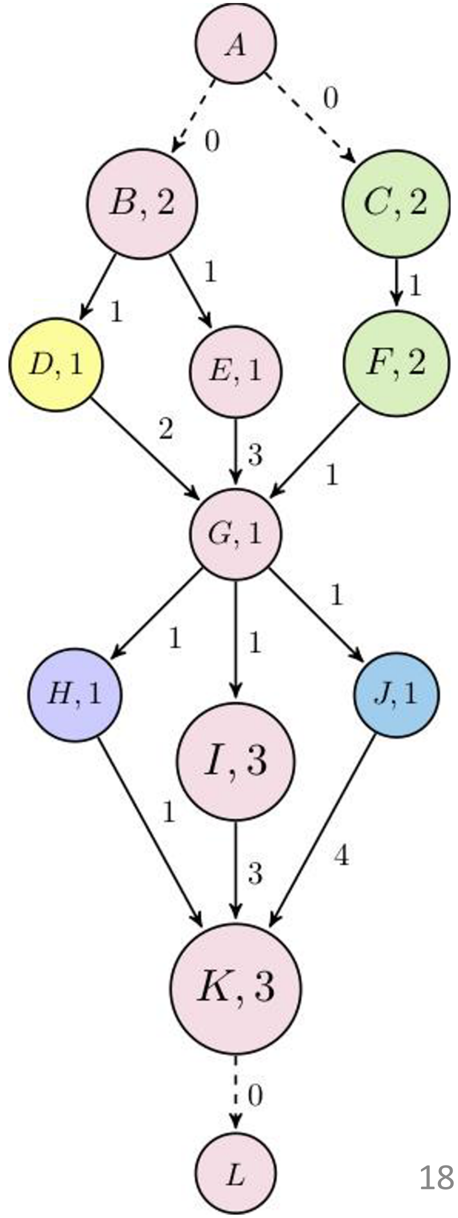


- To achieve both
  - Good quality partitions
  - Reasonable runtime
- Step 1 is divided into 3 stages:
  - Stage 1: Slicing
    - Gets smaller instance representation
    - Obtaining coarser view
    - Capturing costly communications

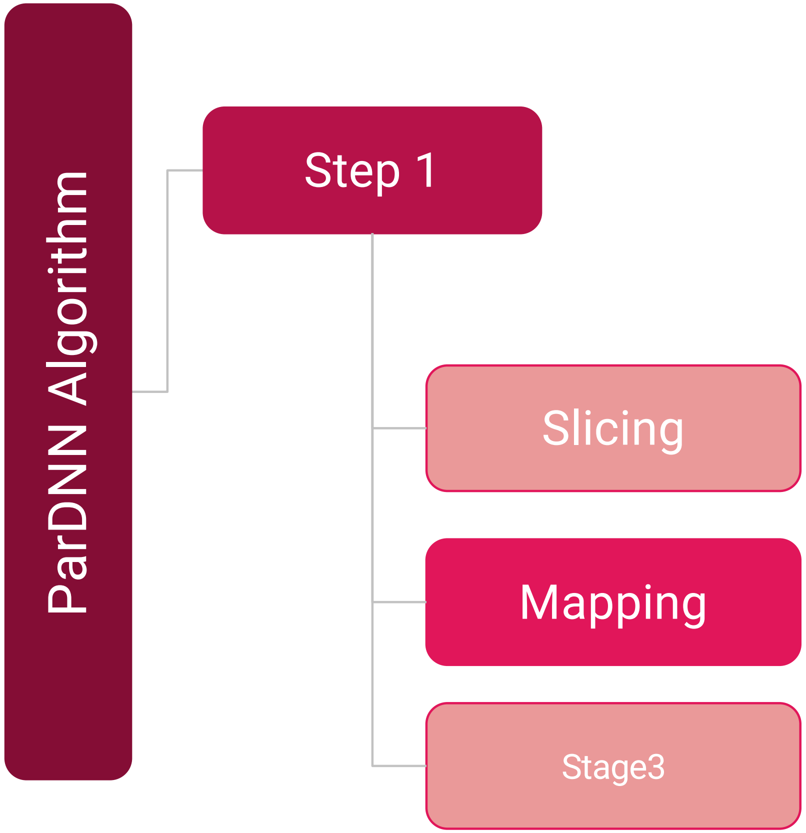
Graph Slicing

- Obtain *K* critical paths of the graph
  - Get the critical path
    - **primary cluster**
  - Remove its nodes & incident edges
- Until the graph has no more nodes
  - Find the heaviest cluster
    - **secondary cluster**
  - Remove its nodes & incident edges

In the figure, pink and green paths are primary clusters
Yellow, blue and purple nodes are secondary clusters

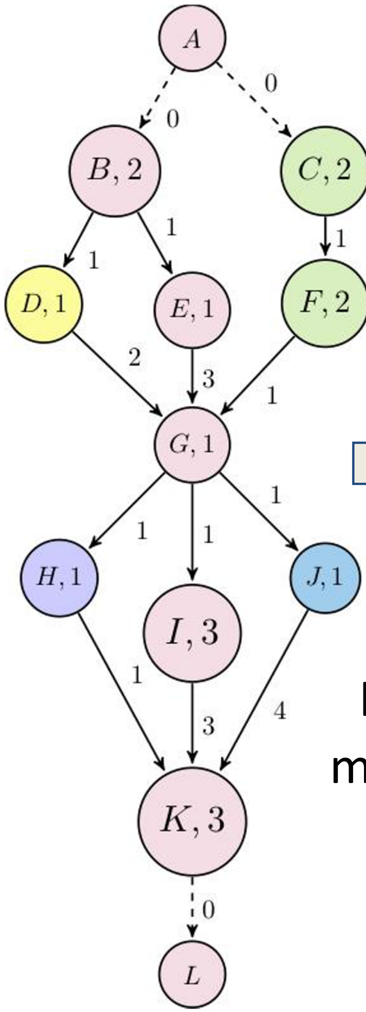


ParDNN Algorithm Overview

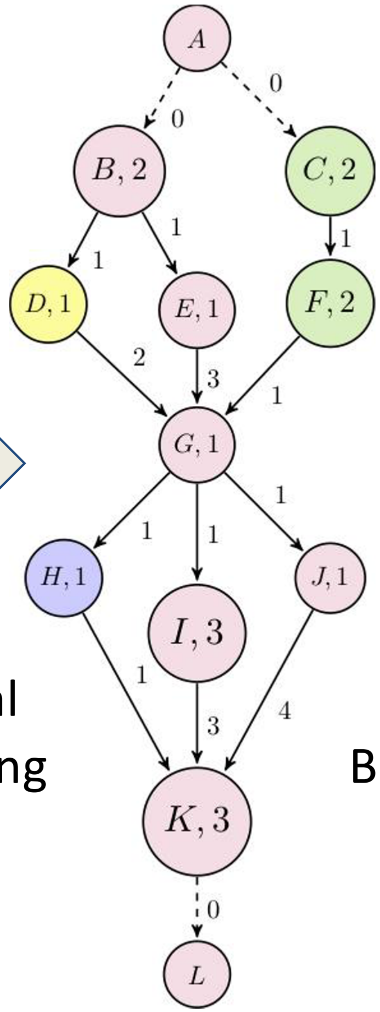


- To achieve both
  - Good quality partitions
  - Reasonable runtime
- Step 1 is divided into 3 stages:
  - Stage 1: Slicing
  - Stage 2: Mapping, **merge secondary clusters with primaries** in a way that:
    - Balances computational loads
    - Minimizes communication

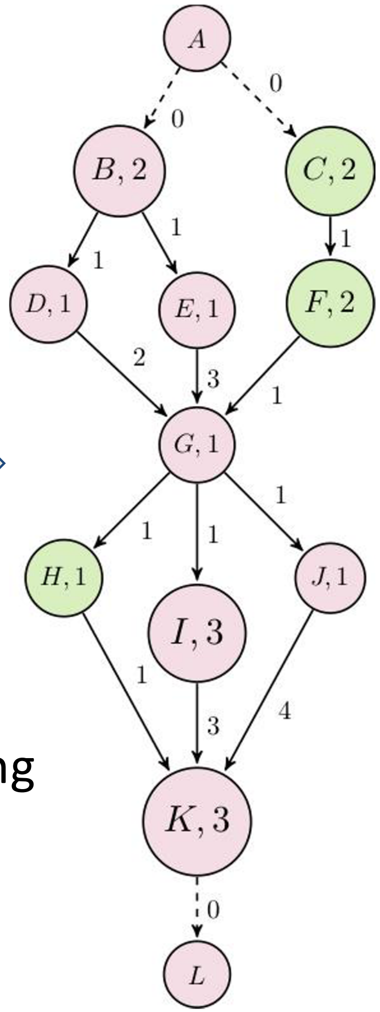
Mapping



Initial merging

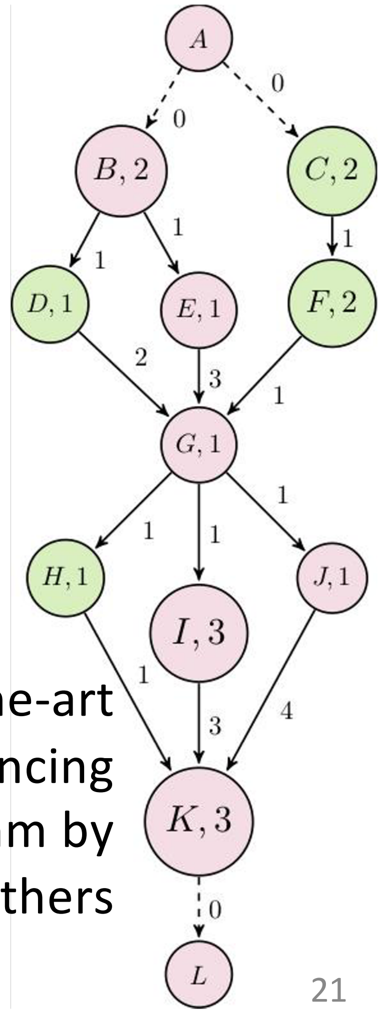


Load Balancing

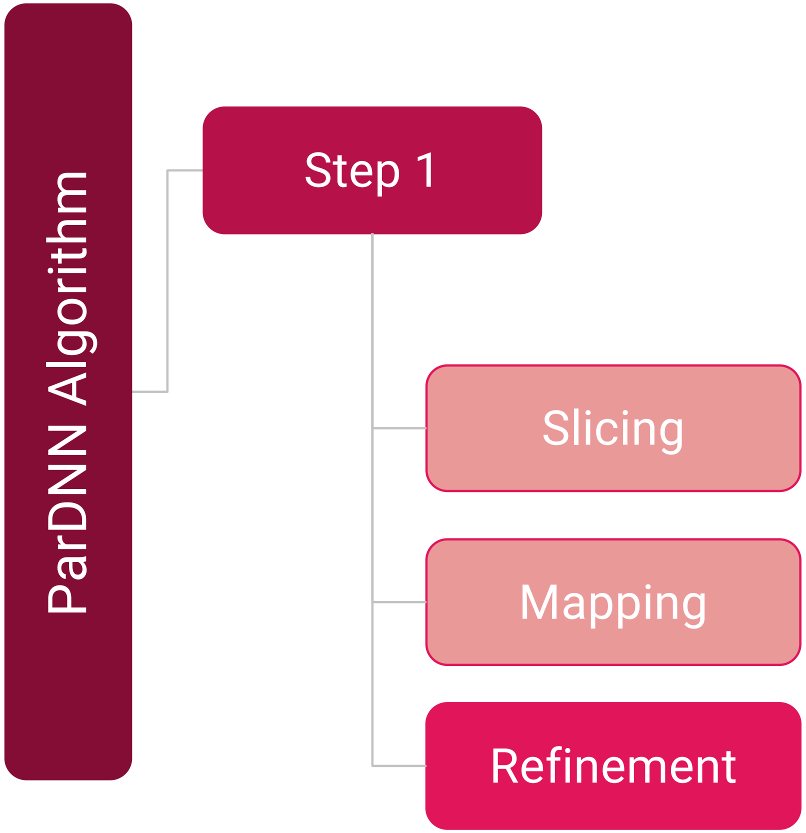


VS

State-of-the-art load balancing algorithm by others

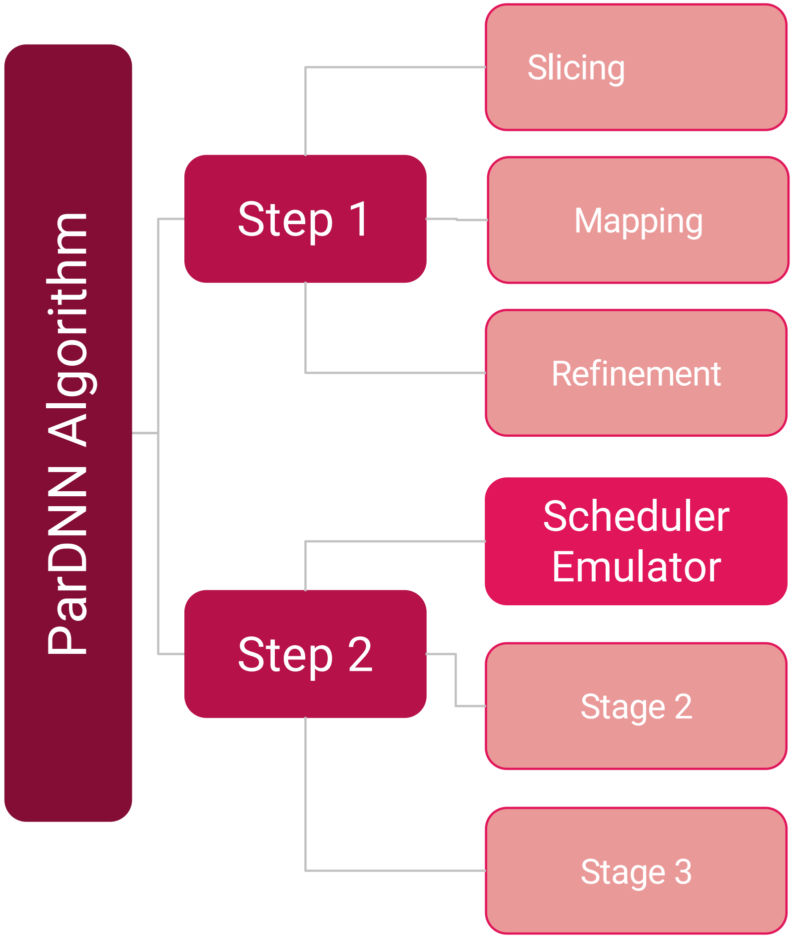


ParDNN Algorithm Overview



- To achieve both
  - Good quality partitions
  - Reasonable runtime
- Step 1 is divided into 3 stages:
  - Stage 1: Slicing
  - Stage 2: Mapping
  - Stage 3: Refinement
    - Enhance partitioning quality
      - At the cluster level
      - At the node level
    - Swap paths and nodes between primaries

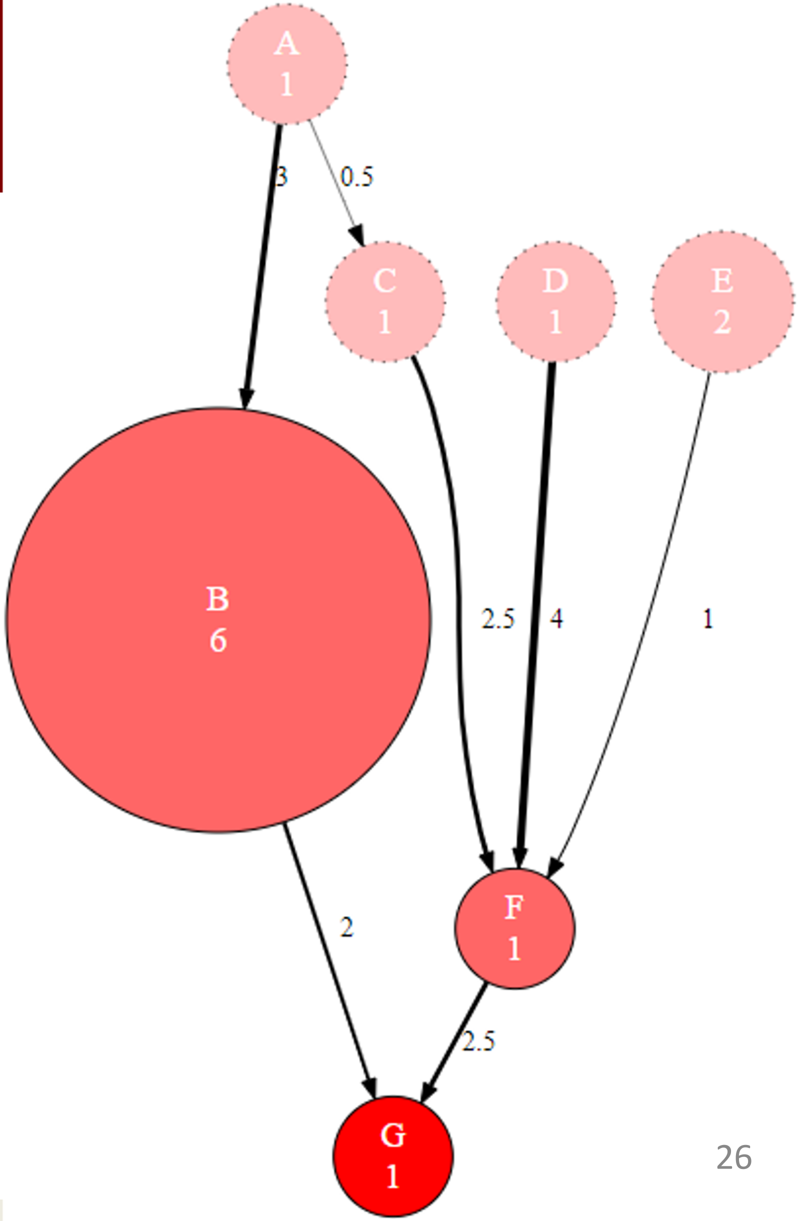
Step 2: Meeting Memory Constraints



- Step 2: Meet the memory consumption constraints
  - Stage 1: Emulate Tensorflow scheduler
    - Get the node's expected **scheduling times**
    - Memory allocation and deallocation patterns

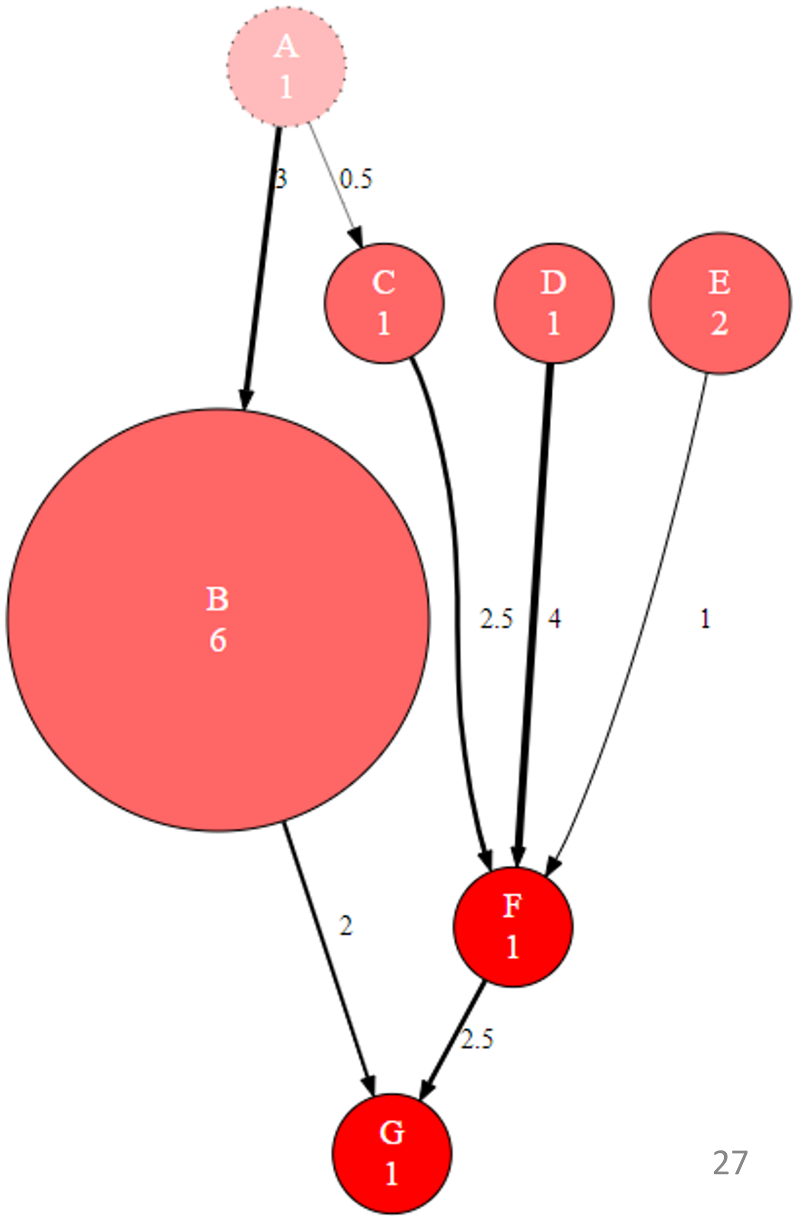
Memory consumption

- Assume a schedule:
  - A, C, D, E, F, B, G.
- Peak memory reserved:
  - 1 + 6 + 1
    - = 8

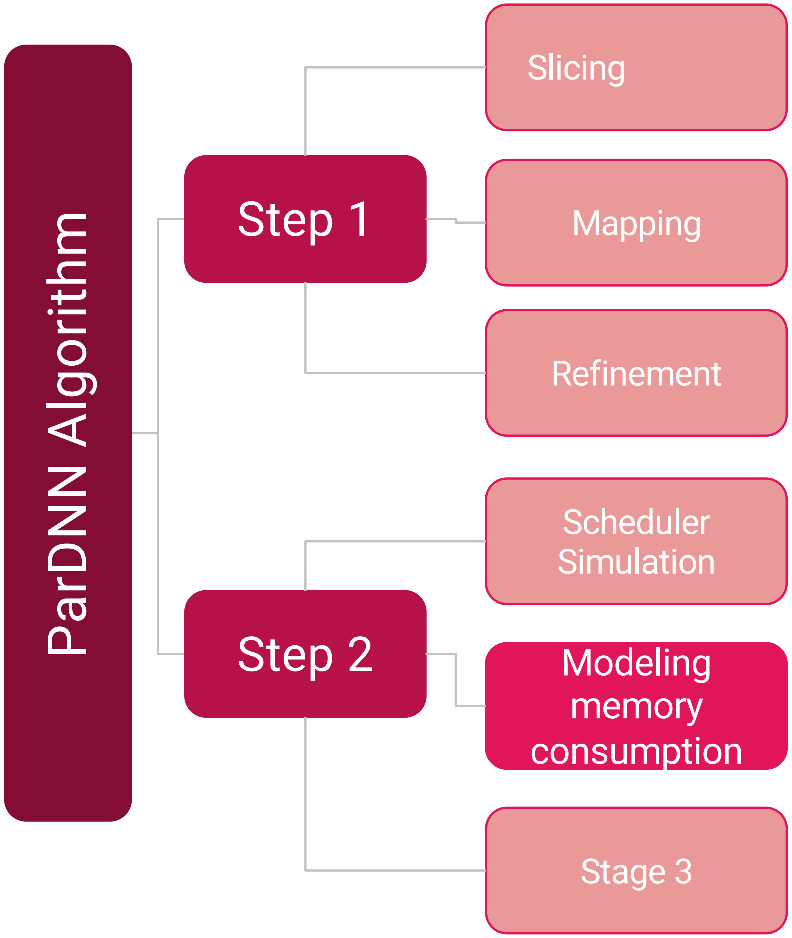


Memory consumption

- Assume **another** schedule:
  - A, B, C, D, E, F, G.
- Peak memory reserved:
  - 6 + 1 + 1 + 2 + 1 = **11**
- It affects as well when having multiple workers.
  - When the data is sent from one to another.

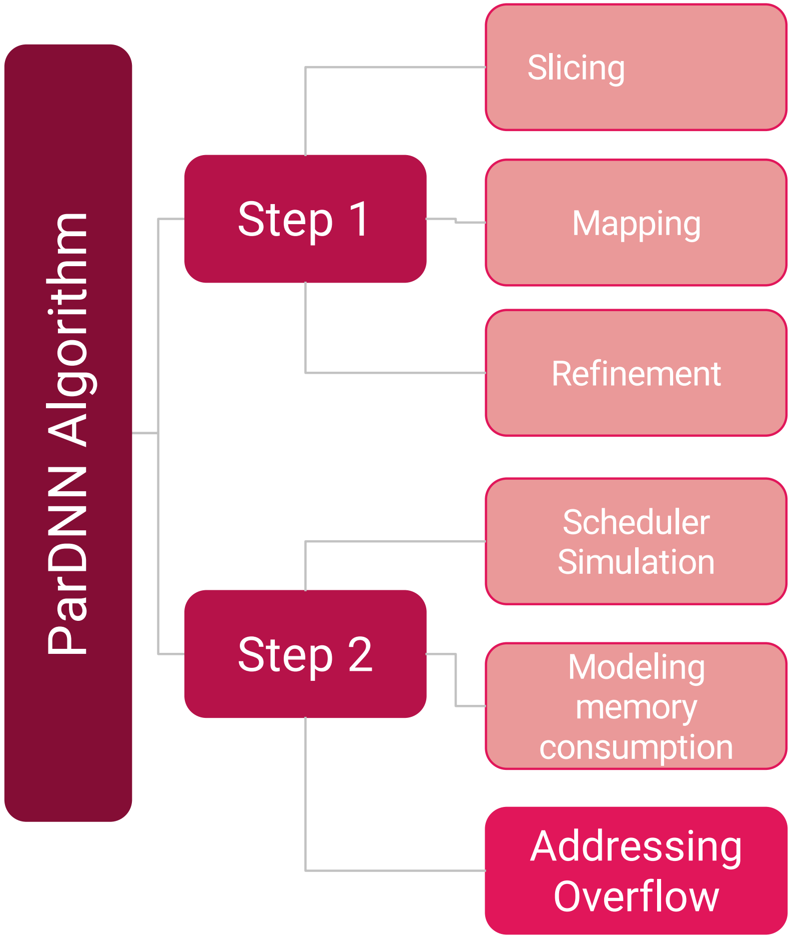


Step2 Stages



- Stage 1: Emulate Tensorflow scheduler
- Stage 2: Modeling memory consumption
  - Derive the memory consumption on a certain device at a certain point in time
  - Calculate memory potentials

Step2 Stages



- Stage 1: Emulate Tensorflow scheduler
- Stage 2: Modeling memory consumption
- Stage 3: Address the memory overflow
  - Which nodes to move?
  - Where to move?

Addressing Memory Overflow

- Each overflow point can be 0-1 min knapsack
  - Move a set of nodes from the overloaded part
    - Summation of their memory potentials at the overflow time ≥ Overflow
  - The cost of a move is how much it affects the existing partitioning:
    - Incur the least possible perturbation on Step 1 results
  - Solved greedily
  - Move the node which , per a memory unit, has the least computation cost and incurs the least communication when moved.

Results

Models and Datasets

- We have experimented with 5 models with 2 different configurations (large and very large)

TABLE III: Specifications of Models Datasets. HSD: Hidden State Dimension, SL: Sequence Length, CHSD: Character Hidden State Dimension, ED: Embedding Dimensions, RU: Residual Units, WF: Widening Factor, FS:Filter Size, MD: Model Dimension

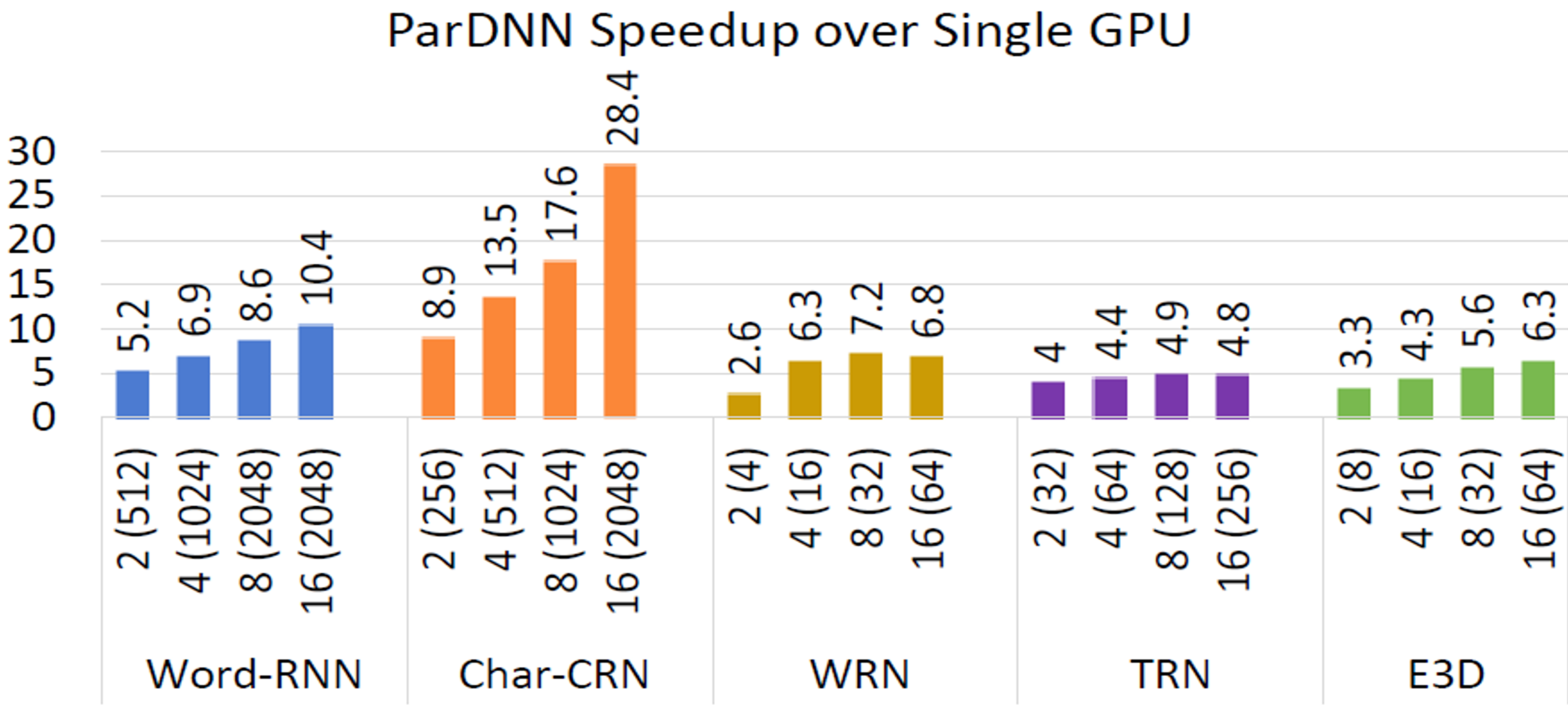
Model	Acronym	#Layers	HSD	SL	#Parameters	#Graph Nodes	Dataset
Recurrent Neural Network for Word-Level Language [51]	Word-RNN	10	2048	28	0.44 billion	11744	Tiny Shakespeare [23]
	Word-RNN-2	8	4096	25	1.28 billion	10578	
		#Layers	CHSD	ED			
Character-Aware Neural Language Models [26]	Char-CRN	8	2048	15	0.23 billion	22748	Penn Treebank (PTB) [33]
	Char-CRN-2	32	2048	15	1.09 billion	86663	
		#Conv. Layers [65]	#RU	WF			
Wide Residual Network [64]	WRN	610	101	14	1.91 billion	187742	CIFAR100 [28]
	WRN-2	304	50	28	3.77 billion	79742	
		#Layers	HSD	MD			
Transformer [54]	TRN	24	5120	2048	1.97 billion	80550	IWSLT 2016 German–English corpus [6]
	TRN-2	48	8192	2048	5.1 billion	160518	
		#Hidden Layers	FS				
Eidetic 3D LSTM[58]	E3D	320	5		0.95 billion	55756	Moving MNIST digits [50]
	E3D-2	512	5		2.4 billion	55756	

Results (Batch size scaling)

	Batch Size Scaling					Increase Over Ideal DP				
Model / #GPUs	1	2	4	8	16	1	2	4	8	16
Word-RNN	16	256	1024	2048	2048	1x	8x	16x	16x	8x
Char-CRN	8	256	512	1024	2048	1x	16x	16x	16x	16x
WRN	1	4	16	32	64	1x	2x	4x	4x	4x
TRN	1	32	64	128	256	1x	16x	16x	16x	16x
E3D	1	4	16	32	64	1x	2x	4x	4x	4x
Word-RNN-2	–	–	32	1024	2048	–	–	1x	16x	16x
Char-CRN-2	–	–	128	512	1024	–	–	1x	2x	2x
WRN-2	–	–	4	16	32	–	–	1x	2x	2x
TRN-2	–	–	3	16	32	–	–	1x	4x	4x
E3D-2	–	–	8	16	32	–	–	1x	1x	1x

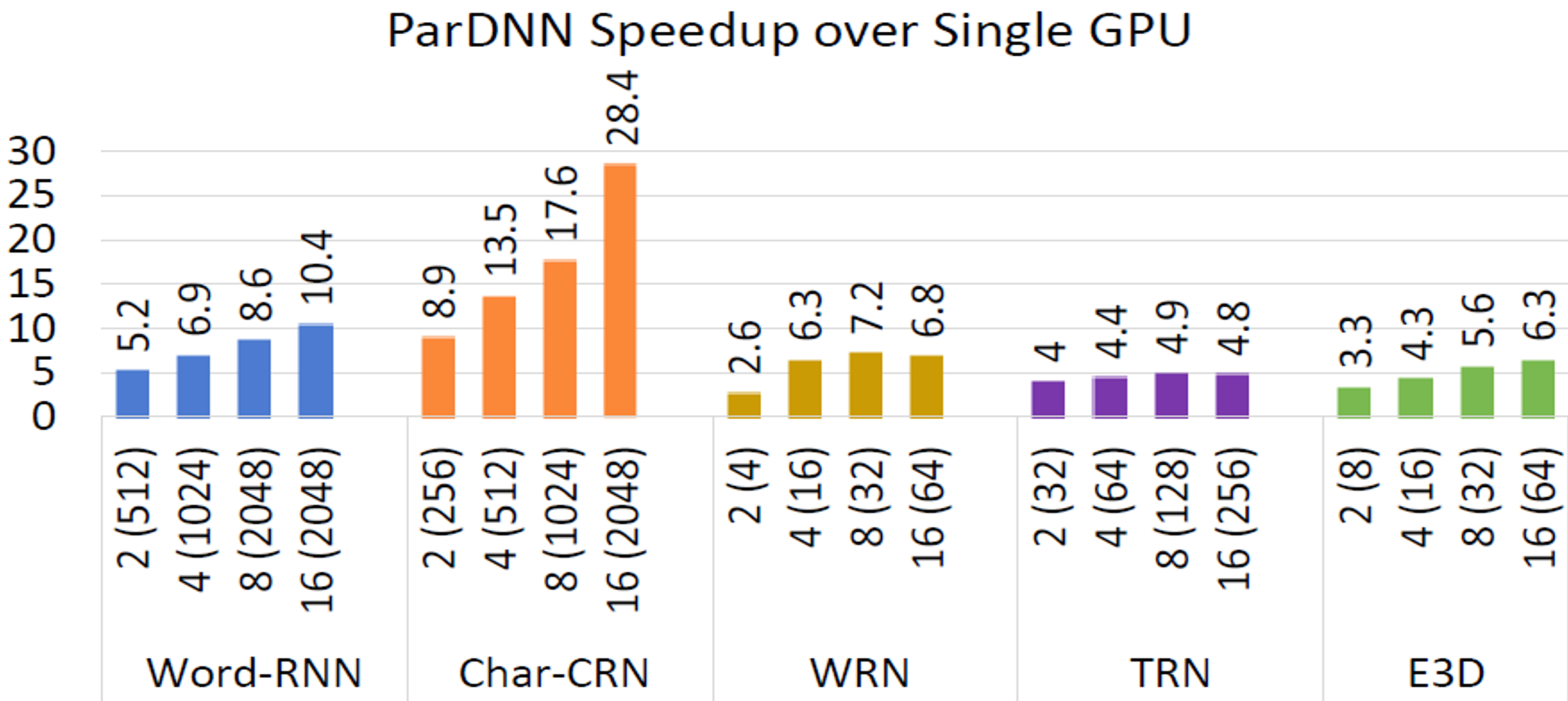
ParDNN enables working with larger data, e.g. pushing larger batches, using certain number of workers.

Results (Training Speedup)



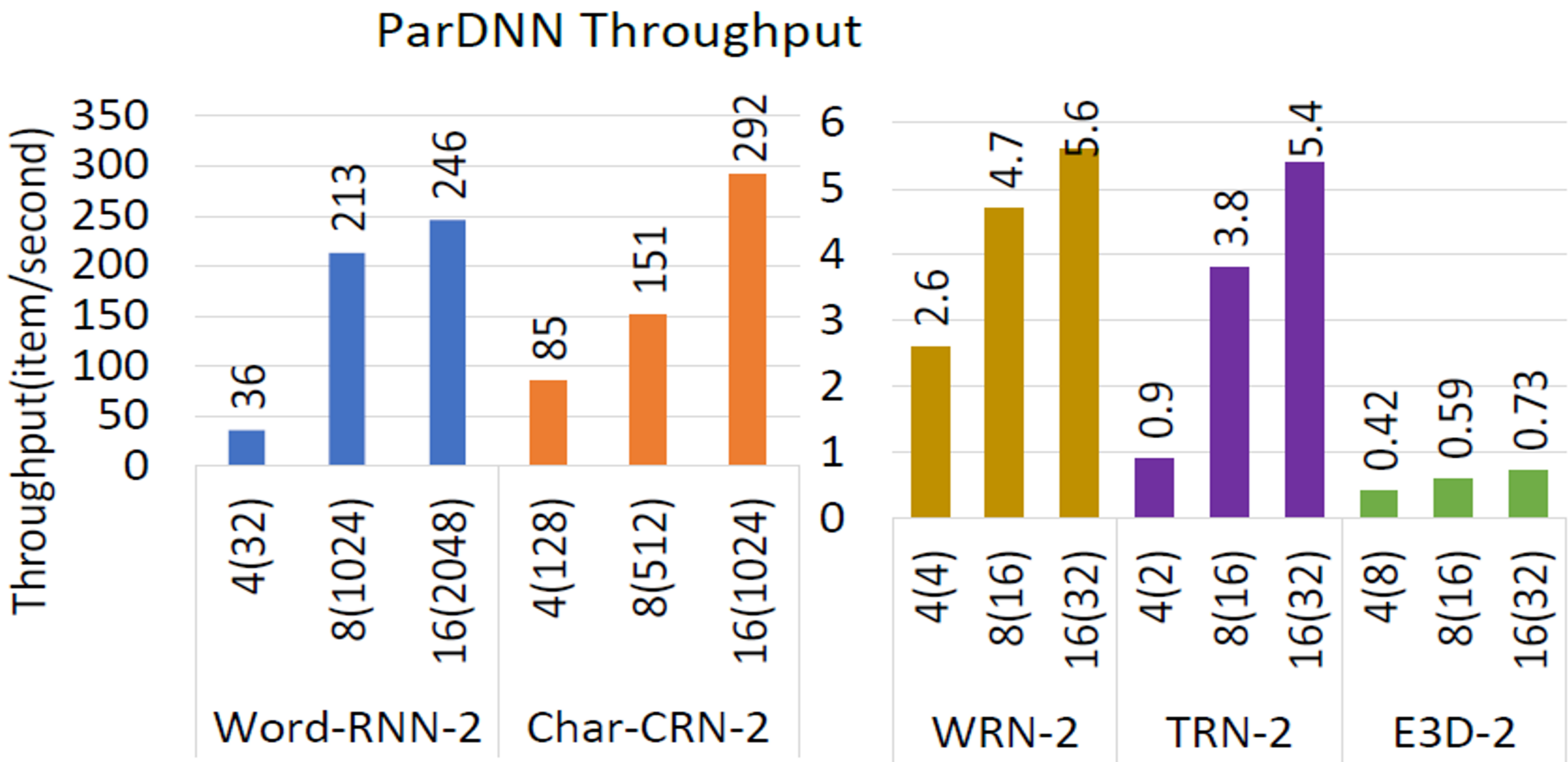
Experimenting with 2, 4, 8, and 16 GPUs. The number in the brackets is the batch size.

Results (Training Speedup)



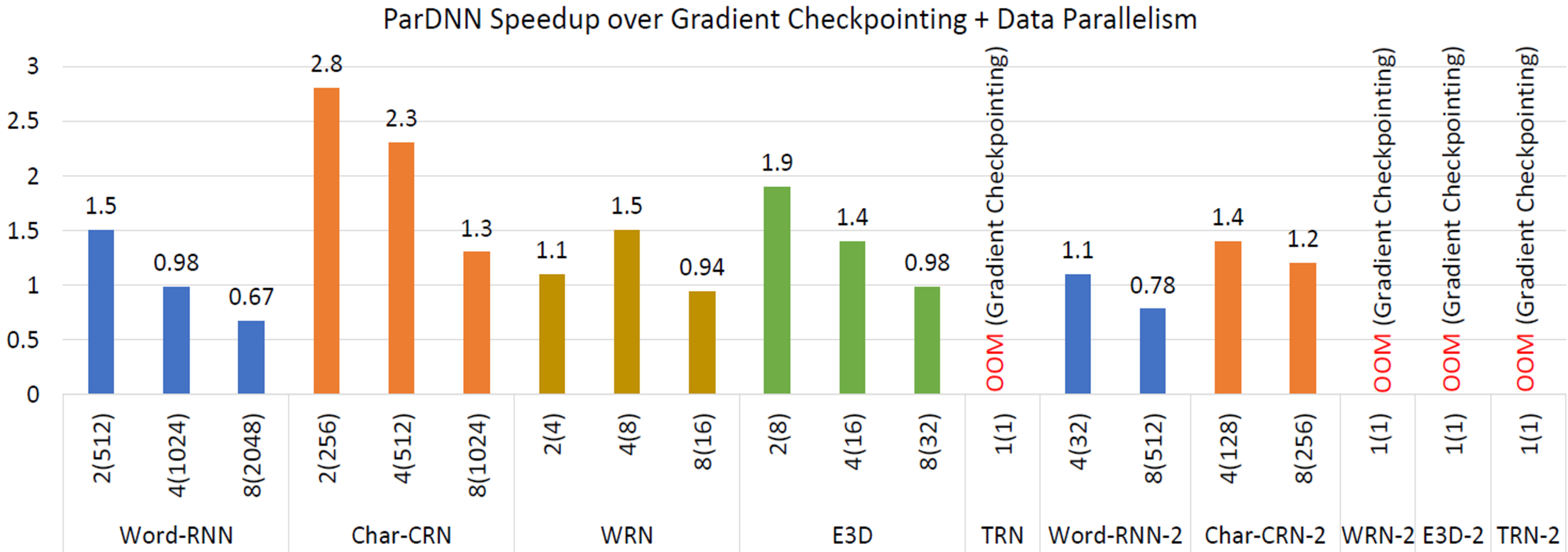
Better resource utilization → Superlinear speedup up to 4 GPUs in all cases.

Results(Larger models scaling)



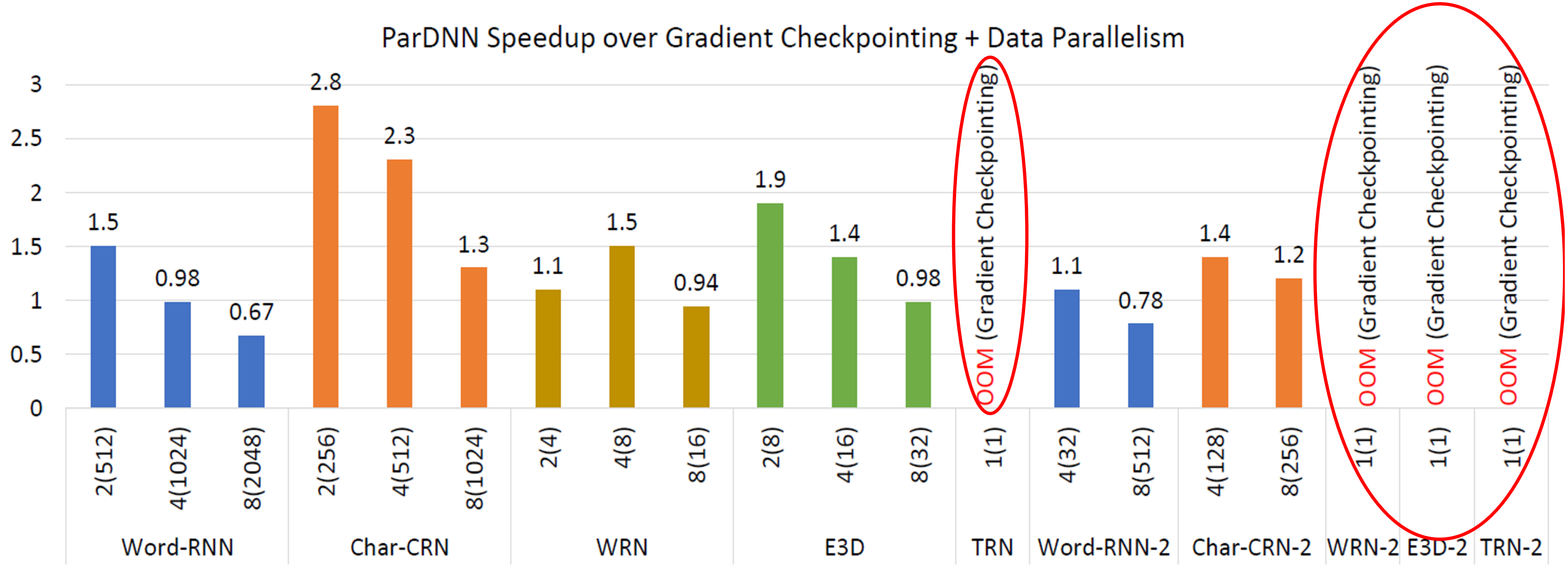
Good scaling with the large models up to 16-GPUs.

Comparison with Gradient Checkpointing + Data parallelism



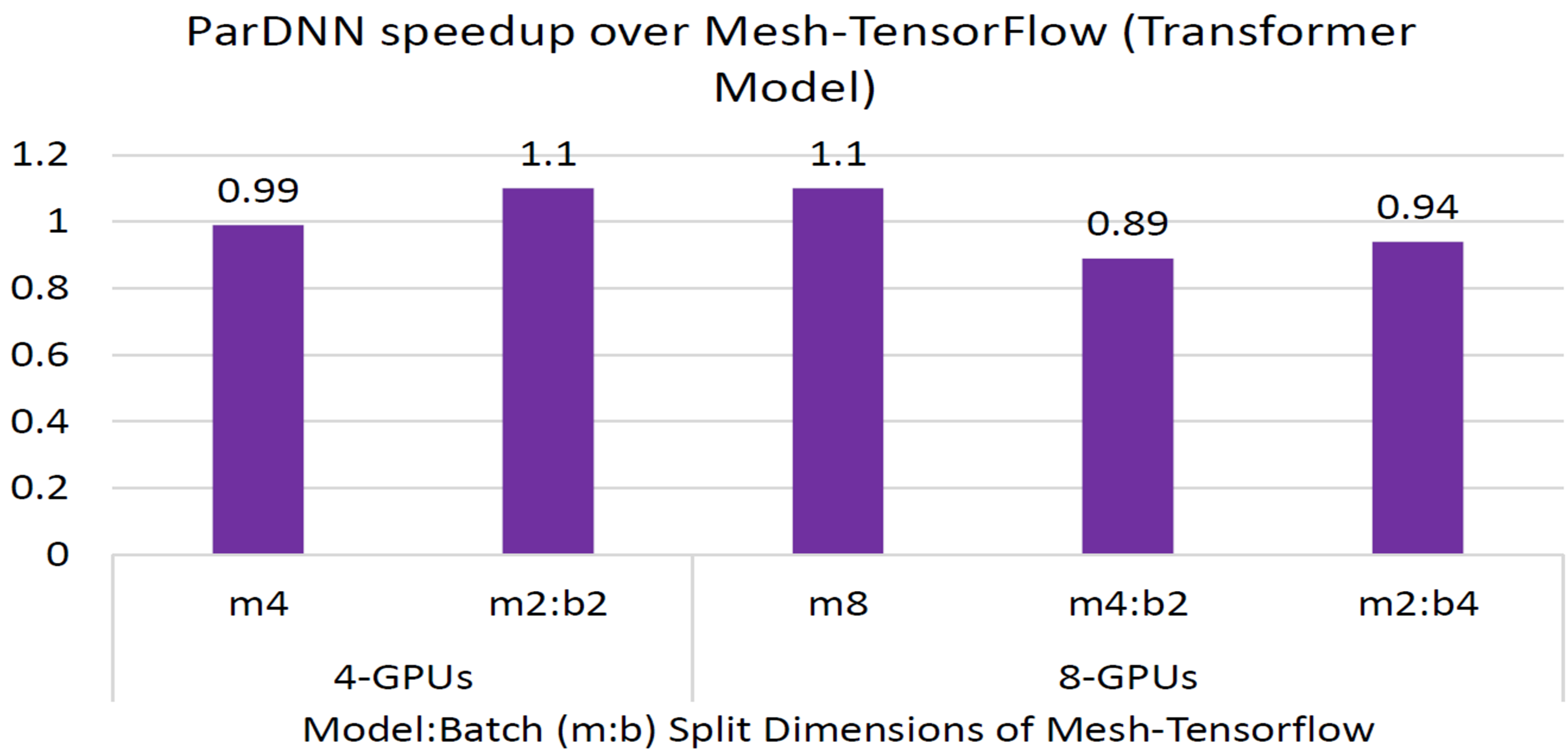
Experimenting with 2, 4, and 8 GPUs. The number in the brackets is the batch size.

Comparison with Gradient Checkpointing + Data parallelism

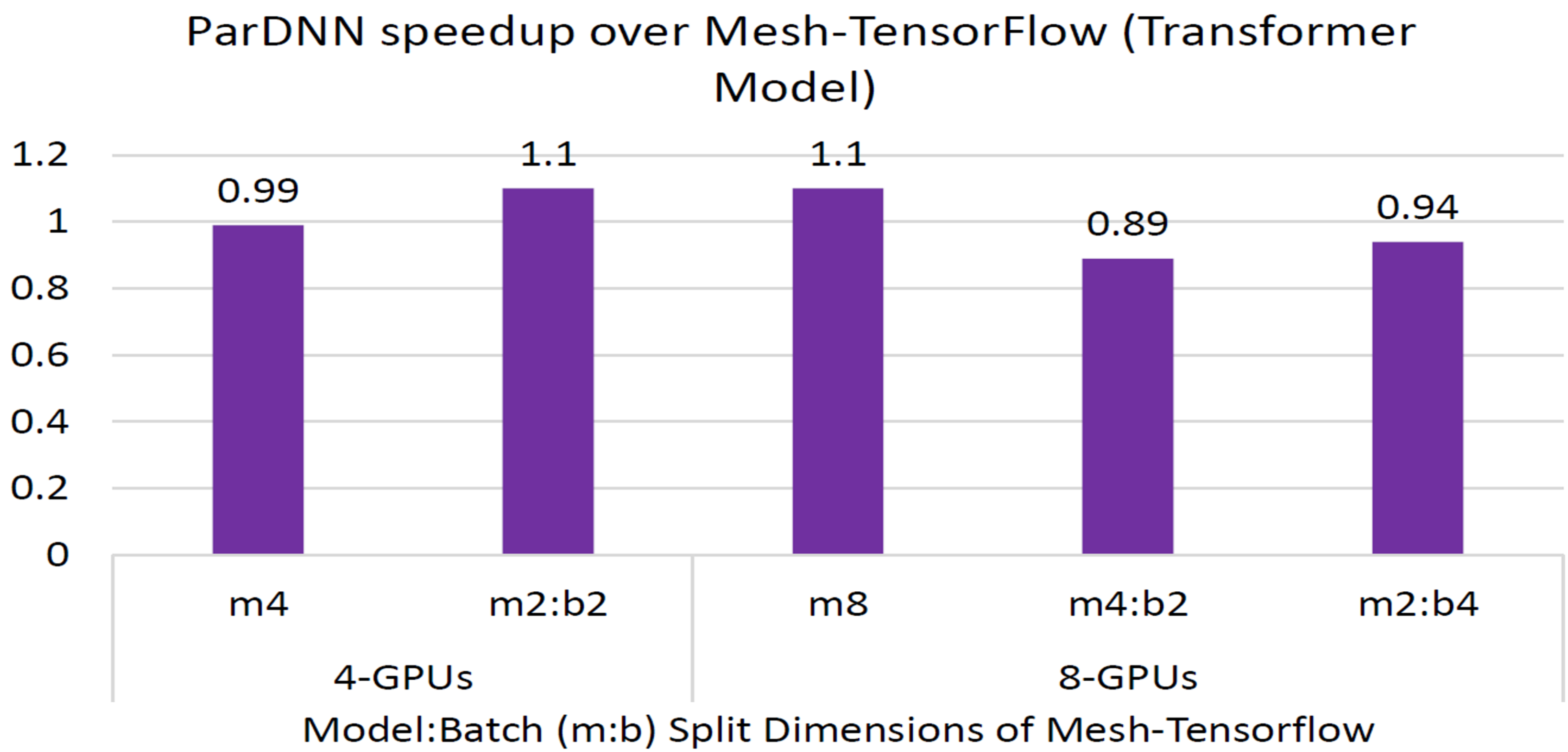


- ParDNN is better in more than half of the cases.
- Checkpointing fails to fit the model 40% of the time.

Comparison with Mesh-Tensorflow



Comparison with Mesh-Tensorflow



- ParDNN automates the partitioning process and needs no programmer intervention and still manages to have similar performance to experts partitioning with Mesh-Tensorflow.

Complexity & Overhead

- The running time of our algorithm in all the experiments ranges from **18 to 117 sec**
- The time complexity of each step as follows:

Step-1	Partitioning to Minimize Makespan
Graph Slicing (inc. sorting)	$O(K(\|V\|+\|E\|))$
Mapping	$O(\|V\|*log\|V\|)$
Refinement	$O(K(\|V\|+\|E\|))$
Step-2	Satisfying Memory Constraints
TensorFlow Scheduler Emulator	$O(\|V\|+\|E\|)$
Tracking Memory Consumption	$O(\|V\|)$
Addressing Overflow	$O(\|V^2\|)$
Overall complexity of PARDNN	$O(\|V\|^2)$

Summary

- We addressed memory constrained DNN models on multiple GPU devices
  - Elegant, non-intrusive and model agnostic approach
  - Two step algorithm design provides efficiency and low overhead
  - Compared to similar approaches, our results are better or provides qualitative advantages
  - Paper is on arxiv: https://arxiv.org/abs/2008.08636

This project is funded by Tübitak 118E801.