Paper Review - Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization[1]

Theodore Long

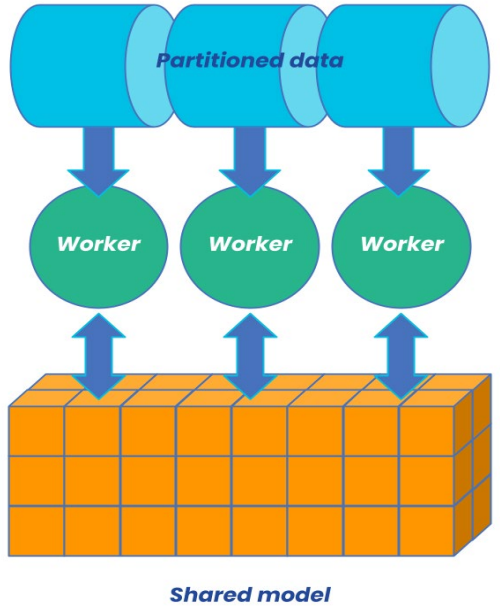
16th November 2022

[1] Authors: Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, Alex Aiken.

Unity: A System for Optimizing DNN Training

- Problem – DNN training is slow and computationally expensive
- Solution – Optimizing computation for faster training and scalability

Algebraic Transformation



Parallelization

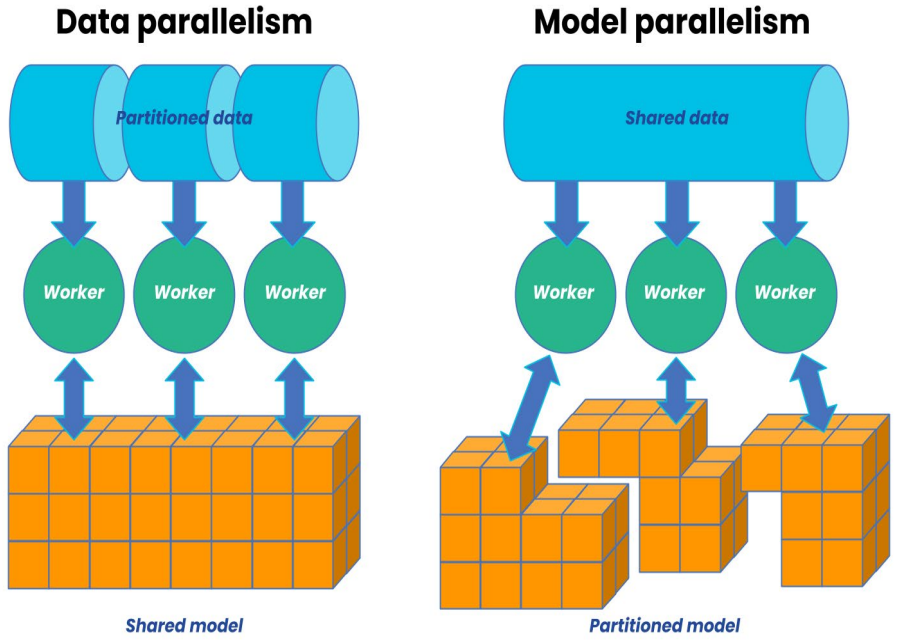
DNN Training Optimizations – A Quick Refresher

- Algebraic Transformation – Changing operator in computation graph to equivalent ones
  - Operator Fusion
  - Depthwise Convolution Reduction
  - … and many more
- Algebraic transformations operate on *logical* computation graph – unaware of device mappings

```
a = np.random.normal(size=[10, 5])
b = np.random.normal(size=[5, 20])
c = np.zeros([10, 20])

# MatMul
for i in range(10):
    for j in range(20):
        for k in range(5):
            c[i][j] += a[i][k] * b[k][j]

# ReLU
for i in range(10):
    for j in range(20):
        c[i][j] = max(0, c[i][j])


# MatMul + ReLU
for i in range(10):
    for j in range(20):
        for k in range(5):
            c[i][j] += a[i][k] * b[k][j]
        c[i][j] = max(0, c[i][j])
```

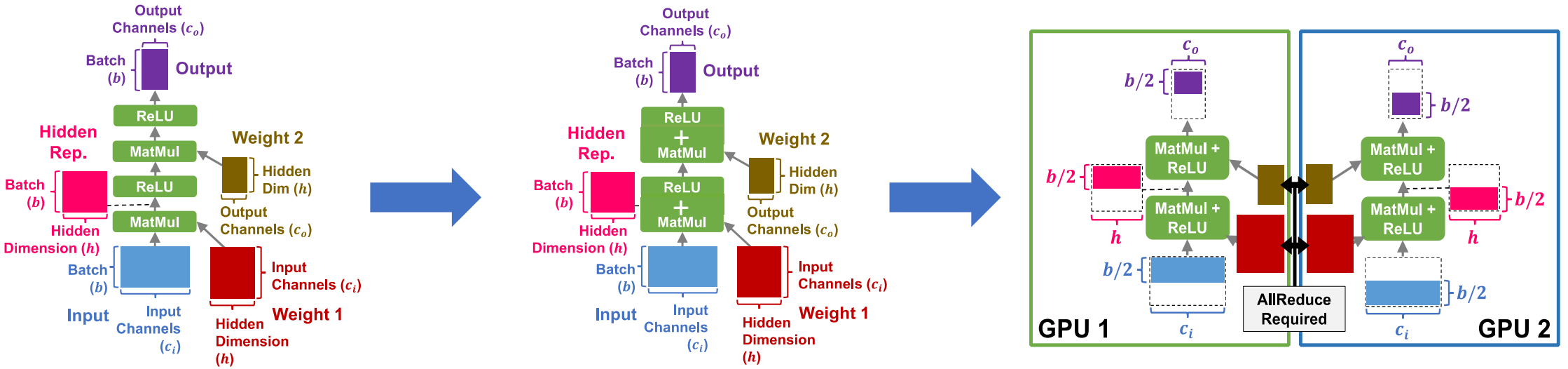
DNN Training Optimizations – A Quick Refresher

- Parallelization – doing computation in parallel across devices
  - Data Parallelism
  - Model Parallelism
  - Spatial Parallelism
  - …
- Parallelization involves tradeoffs – less per-device computation, more communication and synchronization overhead



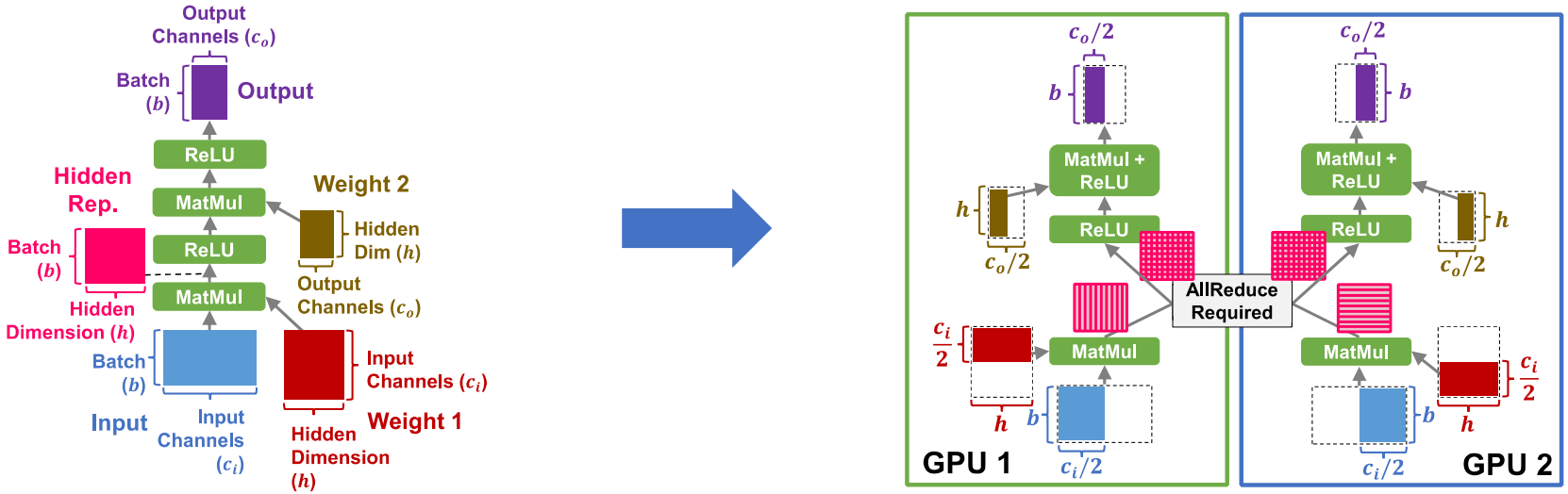
Unity Jointly Optimizes Algebraic Transformations and Parallelization

- Most existing systems focus on *either* algebraic or parallelization optimizations
- For maximum speed up, want to leverage both
- Current approaches only allow sequential optimization



Unity Jointly Optimizes Algebraic Transformations and Parallelization

- Unity performs algebraic and parallelization optimization together
- This allows for better optimizations and performance



Joint Optimization Has Unique Challenges

1. How to represent algebraic and parallel optimization on one computation graph?

2. How to generate hybrid algebraic-parallel optimizations?

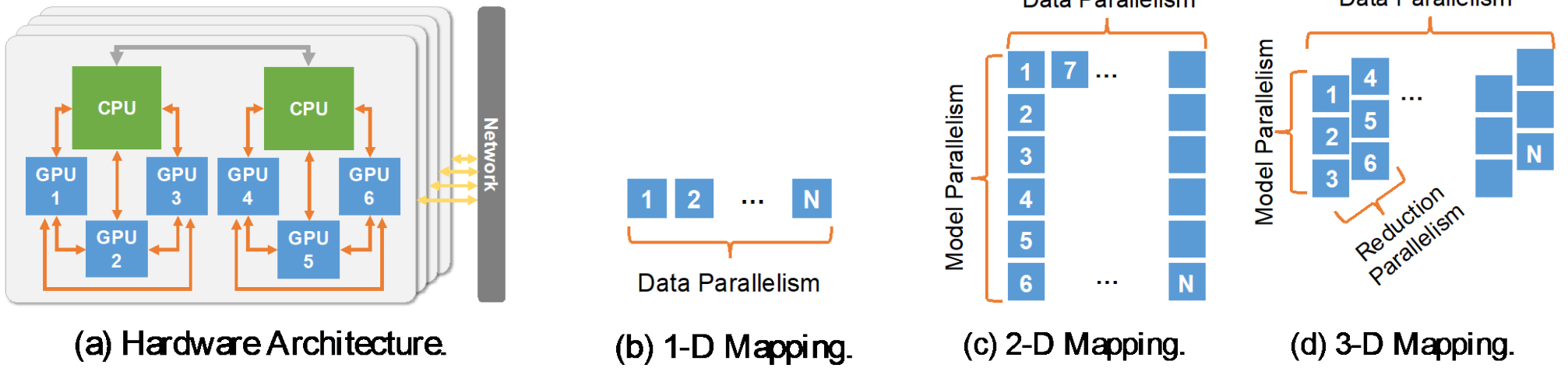
3. How to scale optimization to large models and many devices?

Parallel Computation Graphs allow Joint Representation

- PCG is a computation graph with 2 additional ingredients:
  1. Machine Mapping
  2. Parallelization Operators
- Previous approaches to parallelization *annotate* the computation graph – this is hard to incorporate into joint optimization
- By **directly embedding** parallelism into the PCG, algebraic and parallel transformations are both represented as graph substitutions

Parallel Computation Graphs allow Joint Representation

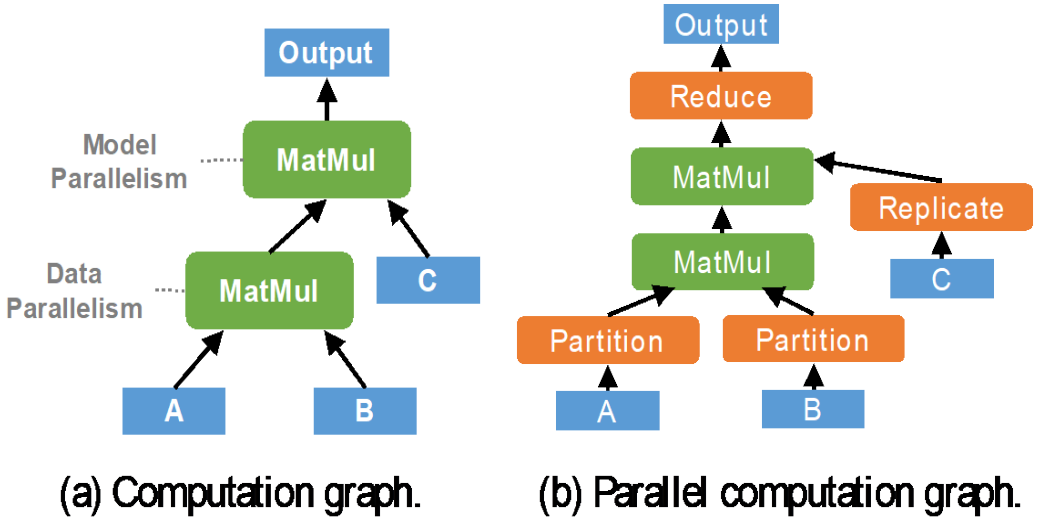
- Every operator has a machine mapping of **tasks** to **devices**
- n-dimensional arrays representing $d_1 \times \ldots \times d_n$ parallel tasks mapped to N devices



(a) Hardware Architecture. (b) 1-D Mapping. (c) 2-D Mapping. (d) 3-D Mapping.

Parallel Computation Graphs allow Joint Representation

- 3 Pairs of parallelization operators
  - Partition and Combine
  - Replicate and Reduce
  - Pipeline and Batch
- Backward pass of one = forward pass of other



(a) Computation graph. (b) Parallel computation graph.

Unity Automatically Generates Valid Substitutions

- Follows TASO superoptimization approach[2] – generate candidates, then formally verify
- Key Idea: We can generate more complex transformation from a small set of ‘basis’ transformations
- Step 1: Generate all possible PCGs up to certain size, and calculate hash of output on standard input Tensors
- Step 2: Formally verify equivalence of all pairs of tensors with same output hash

[2]Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19). Association for Computing Machinery, New York, NY, USA, 47–62.

Unity's Search Algorithm in 3 Steps

- Goal - Given a PCG with machine mappings, find a sequence of substitutions and a final machine mapping which **minimizes per-iteration train time**

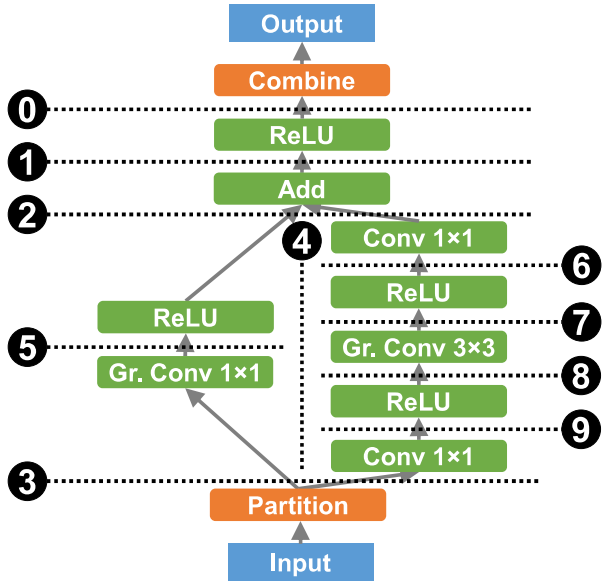
1. Break initial PCG into subgraphs
2. Choose optimal substitutions for each subgraph
   a) Find optimal machine mapping for each substitution
   b) Find optimal substitutions given best machine mappings
3. Recombine optimized subgraphs into final PCG

Substitutions are Selected with a Backtracking Algorithm

- Unity maintains a queue of PCGs sorted by computation time
- While queue is not empty and search budget not exceeded:
  1. Remove best candidate from queue
  2. For each possible substitution:
     - Find optimal machine mapping, evaluate computation time, and add to queue
  3. Remove candidates with time $\geq$ threshold * best time so far (threshold usually set to ~1.05)
- Note – this requires a good computation time estimate!

Unity finds Optimized Machine Mappings through Graph Splits

- Most DNN architectures are composed of parallel chains of sequential computation
- Allows you to decompose PCG with **sequence** and **parallel** graph splits
- For a sequence split $G_1 - n - G_2$, can optimize G_1, n, and G_2 separately
- For each parallel graph split $G_1 \mid G_2$, Unity chooses whether to run G_1 and G_2 sequentially or in parallel
- Unity maintains a cache of optimal mappings



Partitioning the PCG allows for Scalability

- Considering **every** possible substitution for the whole PCG scales as

$$O(2^{\#\text{ of Nodes} \times \#\text{ of Substitutions}})$$

- Instead partition the full PCG into subgraphs of size k (=10 in paper)
- Problem: cannot apply substitutions (e.g. data parallelism) across split points
- Solution: For each split $G_1 - G_2$, consider **all possible partitions** p of split tensor T and optimize under the condition

$$\text{Output partition of } G_1 \;=\; p \;=\; \text{Input partition of } G_2$$

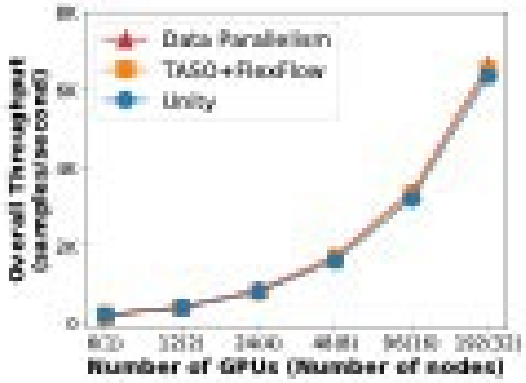
Partitioning the PCG allows for Scalability

- This makes # of possible PCG substitutions to evaluate

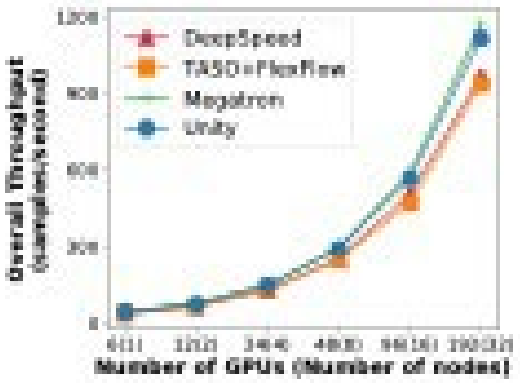
$$O(\frac{g \times \#\text{ of partitions}}{k} \times 2^{k \times \#\text{ of Substitutions}})$$

	All		w/o Split		w/o Cache+Split	
	Time	Scaled	Time	Scaled	Time	Scaled
6 GPUs (1 nodes)	57s	1×	4m 01s	4.3×	37m 01s	38.5×
12 GPUs (2 nodes)	1m 47s	1.9×	11m 15s	16.8×	> 1h	n/a
24 GPUs (4 nodes)	3m 00s	3.1×	> 1h	n/a	> 1h	n/a
48 GPUs (8 nodes)	5m 55s	6.1×	> 1h	n/a	> 1h	n/a

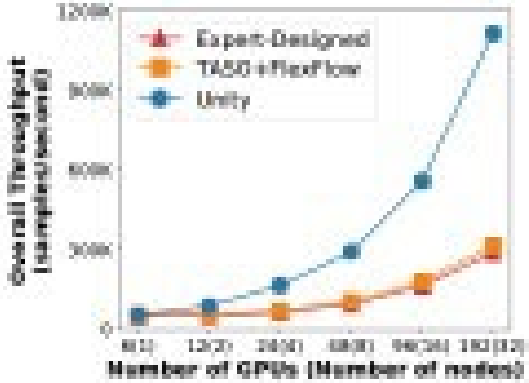
Unity Matches or Beats SOTA Framework Throughputs



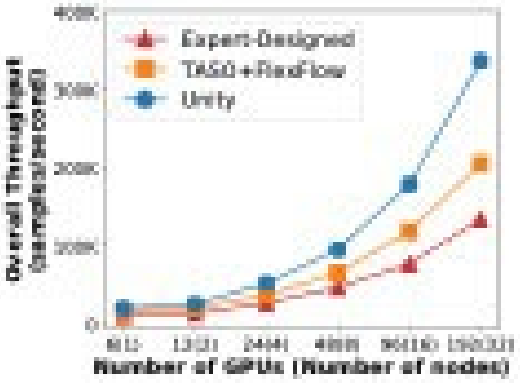
(a) ResNeXt-50.



(b) BERT-Large.



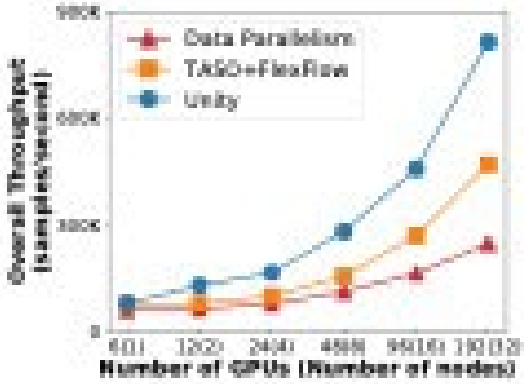
(c) DLRM.



(d) CANDLE-Uno.

(e) Inception-v3.

(f) MLP.



(g) XDL.

Limitations

- Machine mapping generation relies on standard NN structure – some DL architectures do not follow this pattern
- Limited support for pipeline parallelism
- Does not consider non-algebraic or parallelization strategies e.g. rematerialization
- Performance is dependent on good computation time estimates
- Unclear what hardware Unity itself was run on – claim ~20min runtime but not specified what is running this

Conclusion

- Unity combines algebraic and parallel optimizations, enabling hybrid optimizations not achievable in other general-purpose frameworks
- This is enabled by the PCG representation, a powerful abstraction allowing for explicit joint representation and optimization
- System optimizations allow Unity to scale to large models run on 100s of devices
- Unity achieves near-SOTA or SOTA performance on many large models
- Even better performance might be achieved by combining Unity with other approaches e.g. operator optimization with TVM

Questions?