FlexFlow

Beyond Data and Model Parallelism for Deep Neural Networks

Zhihao Jia, Matei Zaharia, Alex Aiken

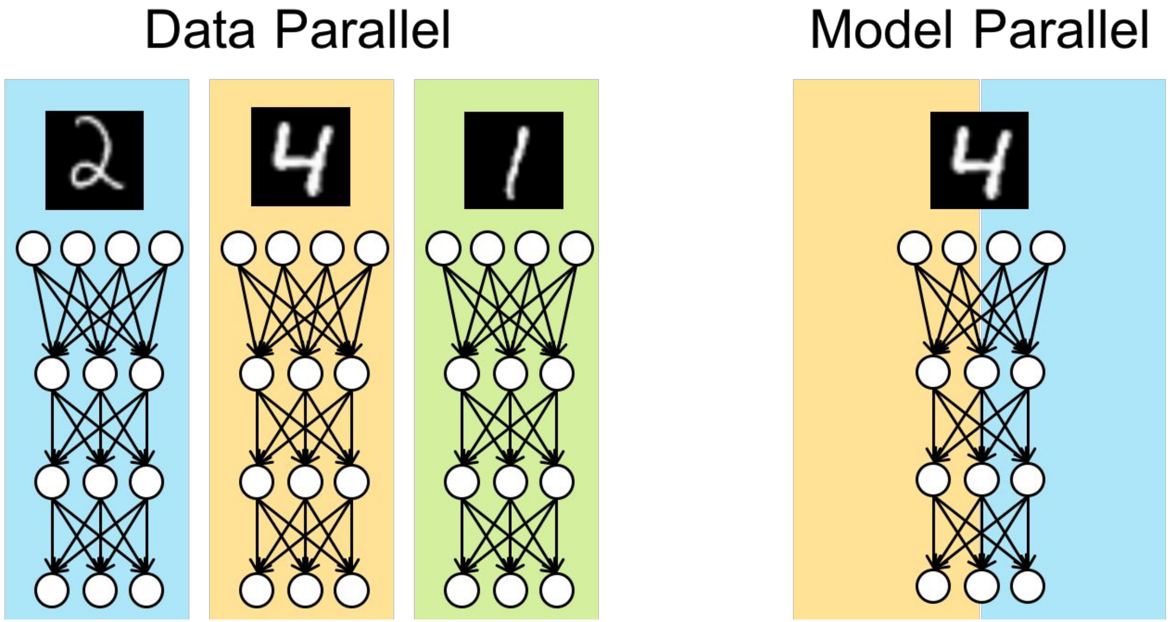
Bowen Chen

Overview

Motivation: Find the best parallel strategy given the **computation graph** and **device topology**.

Key Idea: Define a search space and transforms paralleliation optimization problem into a cost minimization problem.

Previous Work: Data Parallel and Model Parallel



FlexFlow: Beyond Data Parallel and Model Parallel

- Define a larger search space (SOAP) more than data and model parallel.

- A execution simulator that efficiently measure the parallel strategy.

- A search algorithm that find the optimal strategy.

SOAP Search Space

Model the parallelization of an operation o_i by defining how the **ouput tensor** of o_i is partitioned.

- **S**amples -> Data Parallelism
- **O**perators -> Model Parallelism
- *Attributes: partitioning attributes within a sample*
- **P**rameters: partitioning over weights -> Model Parallelism

SOAP Search Space in MatMul -> (S, P)

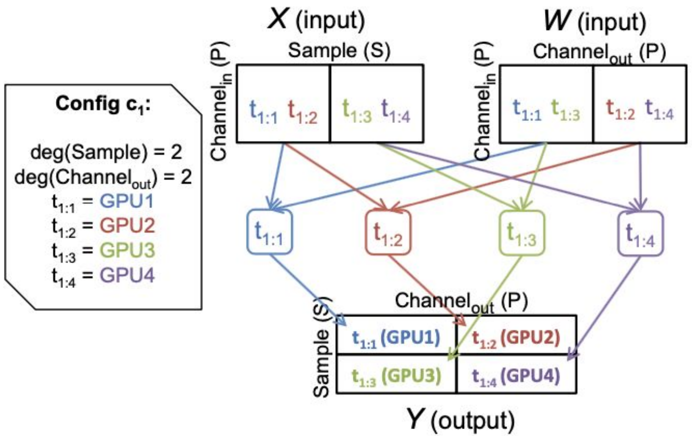


Figure 4: An example parallelization configuration for a matrix multiplication operation.

X W = Y

parallel over parameter (weights)

X [W0 | W1] = [Y0 | Y1]

Task: A computation task needs to be scheduled on a specific device.

SOAP Search Space in 1D-Conv: (S, A, P)

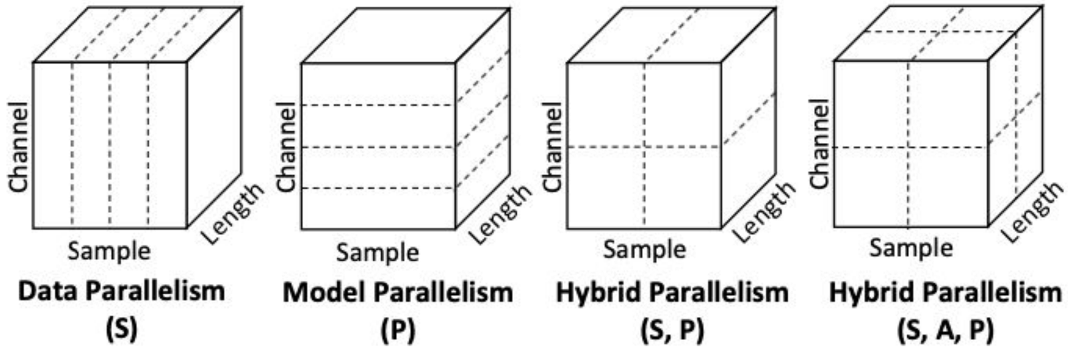


Figure 3: Example parallelization configurations for 1D convolution. Dashed lines show partitioning the tensor.

How to determine which configuration runs faster?

Execution Simulator

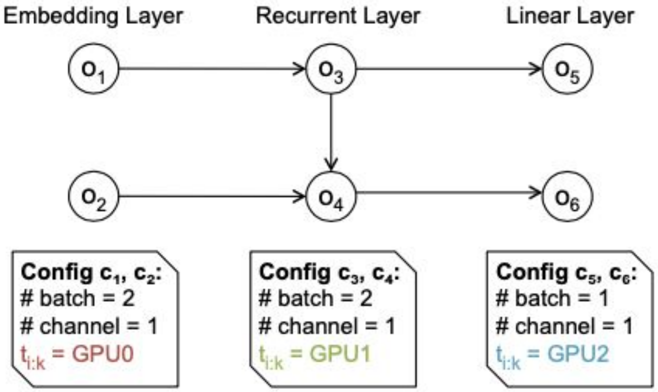
A straight forward idea: Run an iteration on the hardware to measure exectution time.
(requires **12-27 hours** to search for **model parallesim** strategy on 4 GPUS)

An execution simulator (only 14-40s with fewer compuation resources)

- Operation execution time is independent of tensor data content but size
- The connection (communication) between the device can be model as **tensor size / bandwith**
- Device processess assigned task with FIFO scheduling policy
- Runtime has negligible overhead

Execution Simulator: Task Graph

Construct a **task graph** given the **computation graph**, **device layout,** and each operators **configuration**.

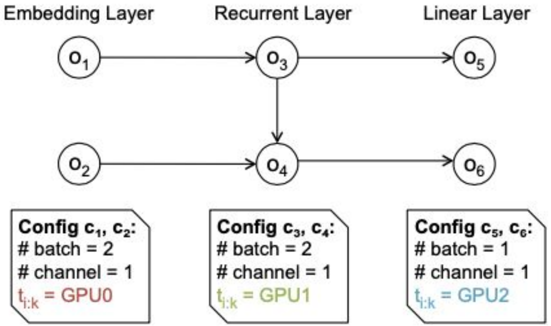


(a) An example parallelization strategy.

op in same layer is assigned to same GPU devices

embedding/Recurrent op apply parition over the Sample

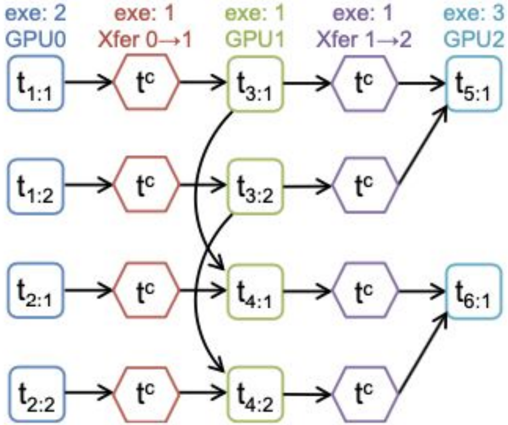
Execution Simulator: Task Graph



(a) An example parallelization strategy.

add computation task

add communication task



(b) The corresponding task graph.

Execution Simulator: Full Simulation

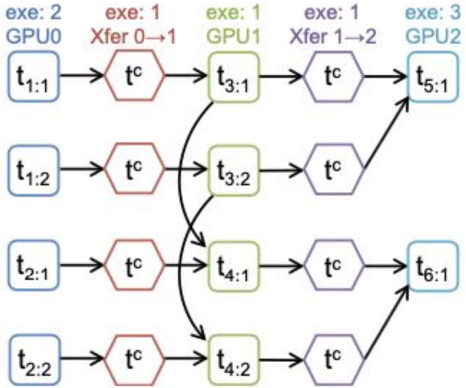
Table 2: Properties for each task in the task graph.

Property	Description
Properties set in graph construction	
`exeTime`	The elapsed time to execute the task.
`device`	The assigned device of the task.
$\mathcal{I}$(t)	$\{t_{in} \mid (t_{in}, t) \in \mathcal{T}_E\}$
$\mathcal{O}$(t)	$\{t_{out} \mid (t, t_{out}) \in \mathcal{T}_E\}$
Properties set in simulation	
`readyTime`	The time when the task is ready to run.
`startTime`	The time when the task starts to run.
`endTime`	The time when the task is completed.
`preTask`	The previous task performed on `device`.
`nextTask`	The next task performed on `device`.
Internal properties used by the full simulation algorithm	
`state`	Current state of the task, which is one of `NOTREADY`, `READY`, and `COMPLETE`.

Algorithm 1 Full Simulation Algorithm.

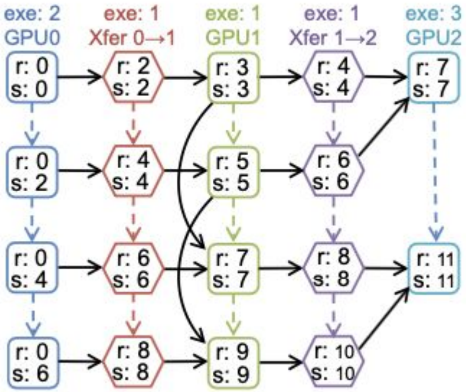
Input: An operator graph $\mathcal{G}$, a device topology $\mathcal{D}$, and a parallelization strategy $\mathcal{S}$.
$\mathcal{T}$ = BUILDTASKGRAPH($\mathcal{G}, \mathcal{D}, \mathcal{S}$)
readyQueue = {} *// a priority queue sorted by readyTime*
for t $\in \mathcal{T}_N$ **do**
    t.state = `NOTREADY`
    if $\mathcal{I}(\mathrm{t}) = \{\}$ **then**
        t.state = `READY`
        readyQueue.enqueue(t)
while readyQueue $\neq \{\}$ **do**
    Task t = readyQueue.dequeue()
    Device d = t.device
    t.state = `COMPLETE`
    t.startTime = max{t.readyTime, d.last.endTime}
    t.endTime = t.startTime + t.exeTime
    d.last = t
    for n $\in \mathcal{O}(\mathrm{t})$ **do**
        n.readyTime = max{n.readyTime, t.endTime}
        if all tasks in $\mathcal{I}(\mathrm{n})$ are `COMPLETE` **then**
            n.state = `READY`
            readyQueue.enqueue(n)
return max{t.endTime | t $\in \mathcal{T}_N$}

Execution Simulator: Full Simulation



(b) The corresponding task graph.

full simulation given the properties



(c) The task graph after the full simulation algorithm.

Execution Optimizer: MCMC sampling

Given a large space of parallelization strategies, and their estimated execution time from simulator, how to find the best strategy efficiently?

MCMC sampling: obtain samples from a probability distribution where sample with higher probability distribution is visited more often.

Execution Optimizer: MCMC sampling

Model the p(S) based on the cost from the simulator

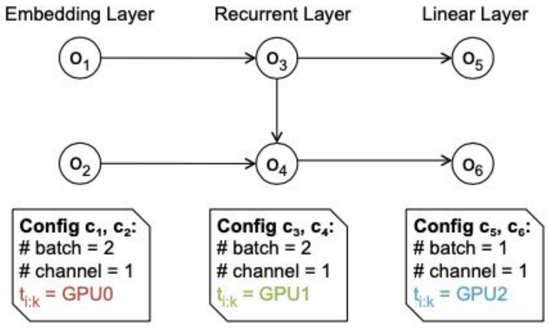
$$p(\mathcal{S}) \propto \exp\left(-\beta \cdot cost(\mathcal{S})\right)$$

Modify **a single operator's configuration** of **S** and genegrate **S***

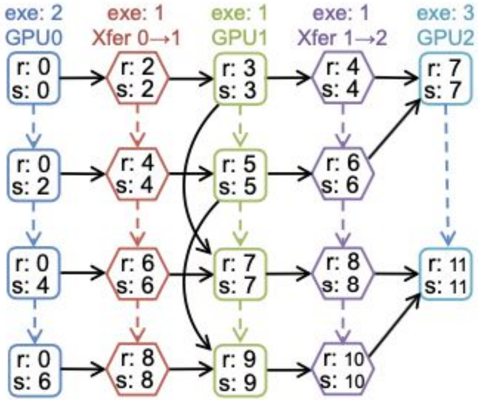
$$\alpha(\mathcal{S} \to \mathcal{S}^*) = \min\left(1, p(\mathcal{S}^*)/p(\mathcal{S})\right)$$
$$= \min\Big(1, \exp\left(\beta \cdot (cost(\mathcal{S}) - cost(\mathcal{S}^*))\right)\Big)$$

End criteria: With an initial S, the search is done when search budget is exhausted or no further improvement in half of search time.

Execution Optimizer: MCMC sampling



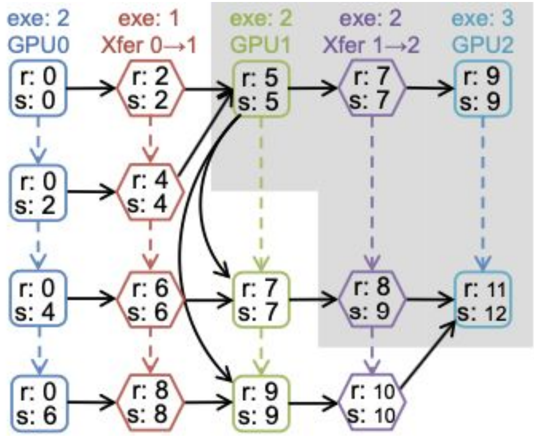
(a) An example parallelization strategy.



(c) The task graph after the full simulation algorithm.

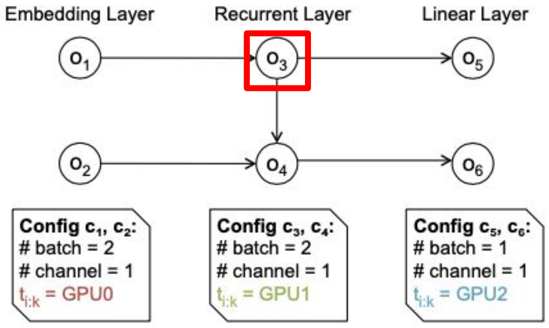
Which operator's configuration

is changed and how?

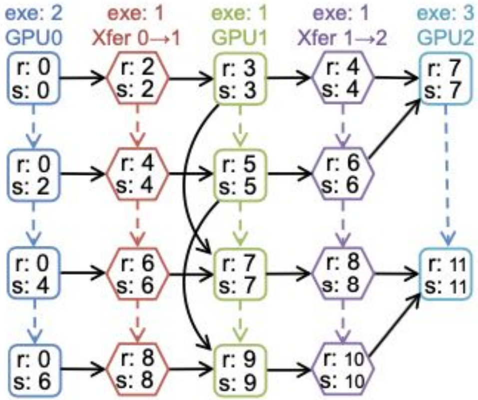


(d) The task graph after the delta simulation algorithm.

Execution Optimizer: MCMC sampling

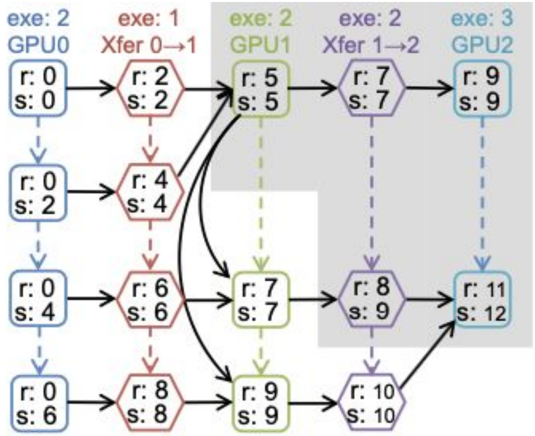


(a) An example parallelization strategy.



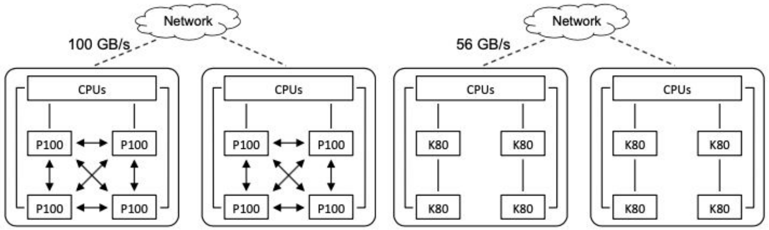
(c) The task graph after the full simulation algorithm.

o_3 **# batch=2->1**



(d) The task graph after the delta simulation algorithm.

Evaluation



(a) The P100 Cluster (4 nodes). (b) The K80 Cluster (16 nodes).

Figure 6: Architectures of the GPU clusters used in the experiments. An arrow line indicates a NVLink connection. A solid line is a PCI-e connection. Dashed lines are Infiniband connections across different nodes.

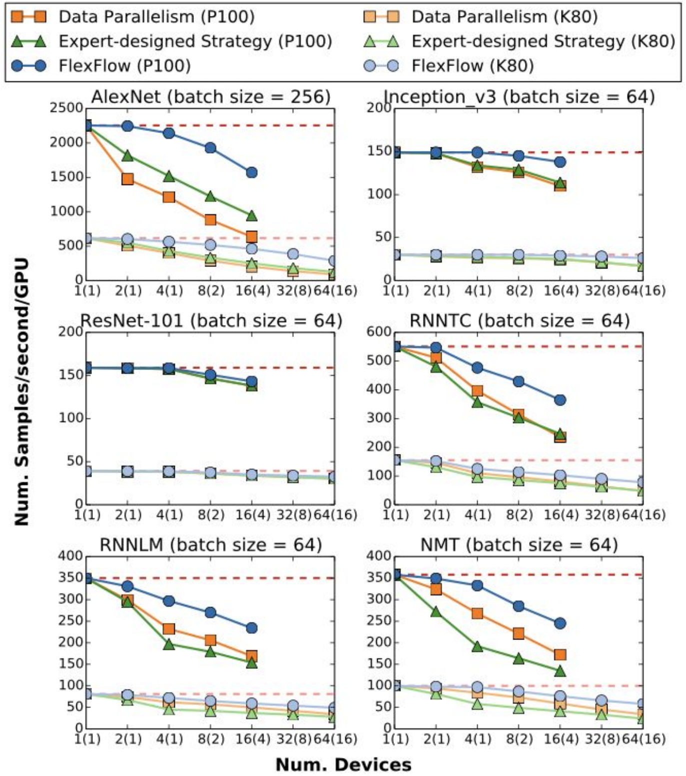


Figure 7: Per-iteration training performance on six DNN benchmarks. Numbers in parenthesis are the number of compute nodes used in the experiments. The dash lines show the ideal training throughput.

Discussion Problems

- Flexflow only considers equal size partitions in each dimension for load-balance. Considering the cluster with heterogeneous devices, can we have uneven splits? Will that leads to any issue?

- How to extend FlexFlow to handle concurrent computation tasks on the same cluster?

- Considering the operator has to do with randomness, i.e. dropout/randperm, is it still possible to parallelize over the Attribute Dimension?

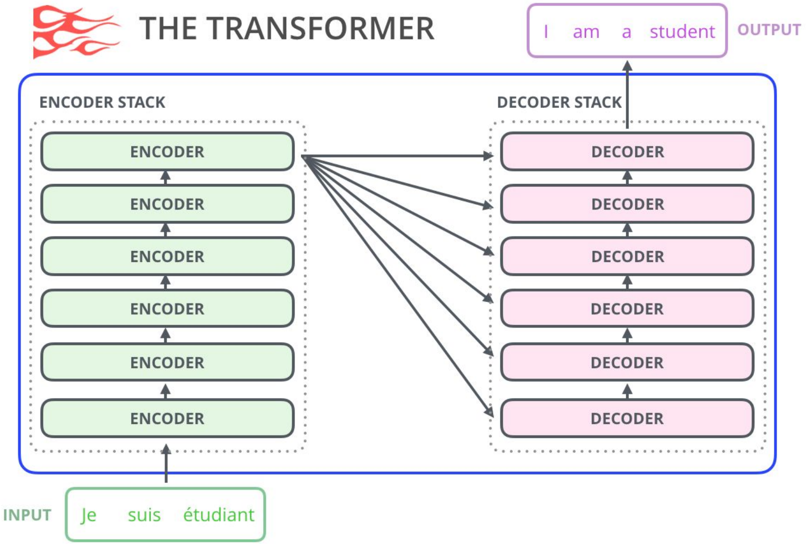
Carnegie Mellon University

GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding

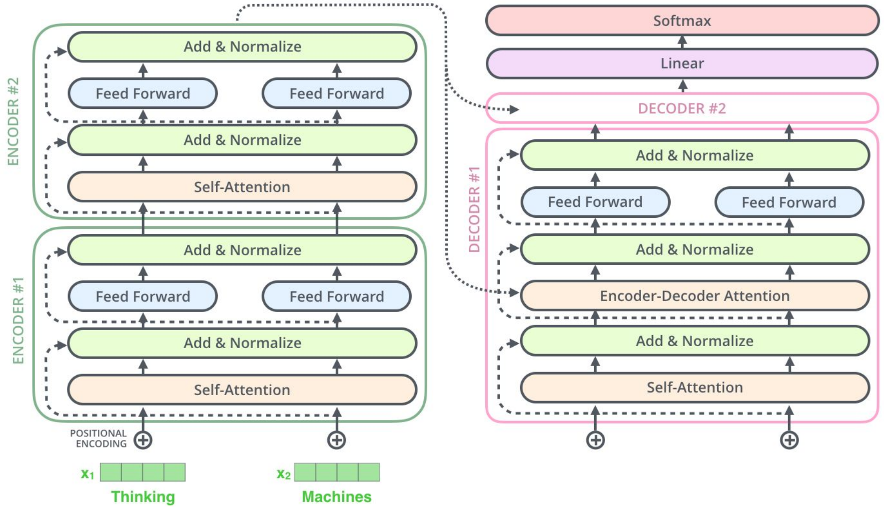
Authors: Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, Zhifeng Chen

Presenter: Jiajun Wan, Date: 02/21/2022

Background

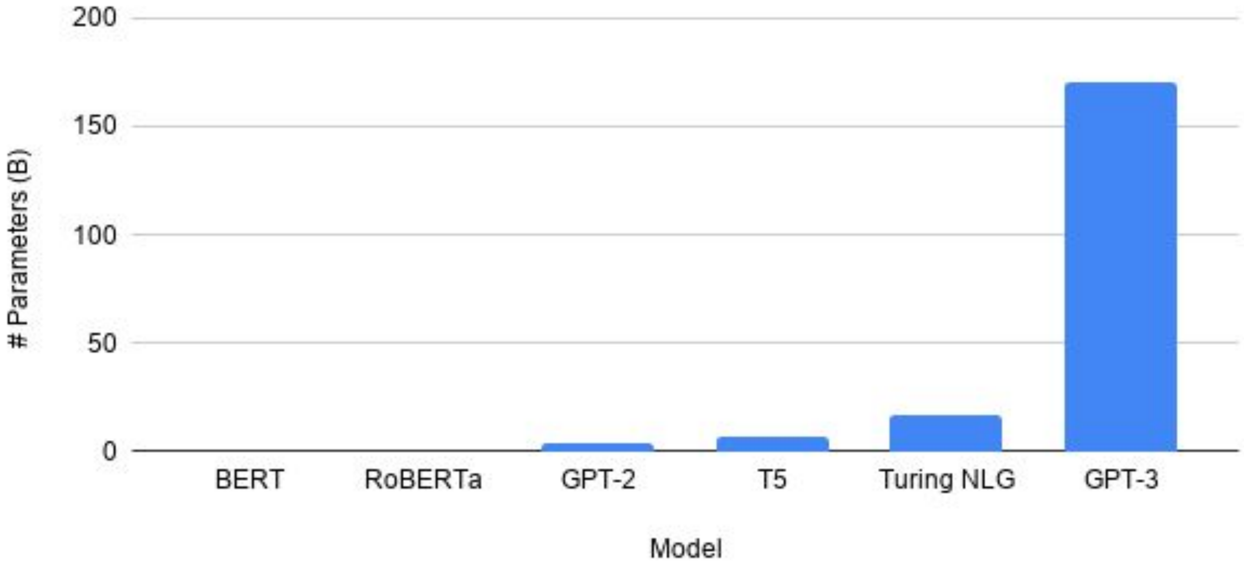


THE TRANSFORMER
I am a student
OUTPUT
ENCODER STACK
DECODER STACK
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
DECODER
DECODER
DECODER
DECODER
DECODER
DECODER
INPUT
Je suis étudiant



ENCODER #2
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
POSITIONAL ENCODING
x1
x2
Thinking
Machines
Softmax
Linear
DECODER #2
DECODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Encoder-Decoder Attention
Add & Normalize
Self-Attention

Background

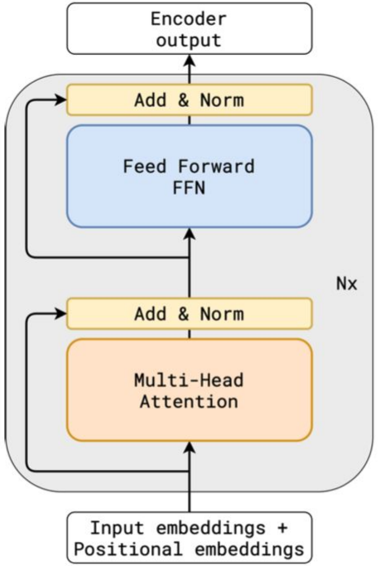


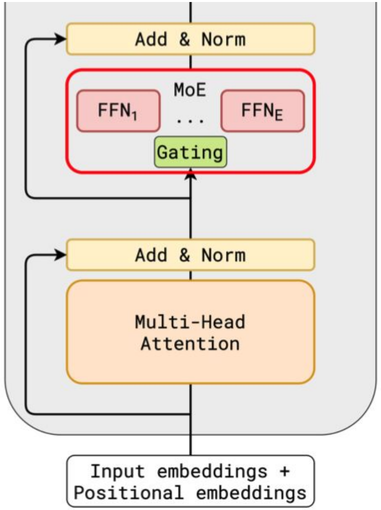
Carnegie Mellon University

Scale Up

- Make it DEEP, go vertically

- Make it WIDE, go horizontally





Carnegie Mellon University

Challenges

- Computation Cost
  - Scale efficiently
- Ease of programming
  - Abstraction and lightweight API
- Efficient implementation on parallel devices
  - Extension to compiler such as XLA

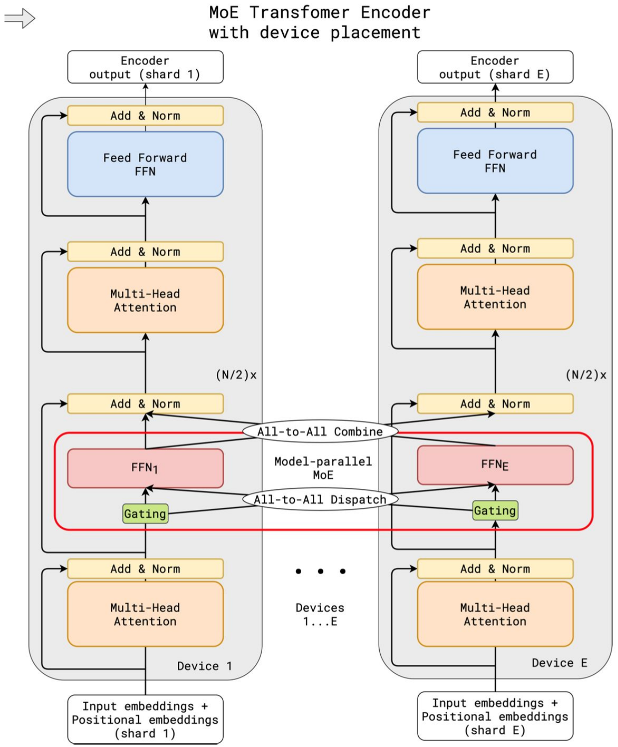
Carnegie Mellon University

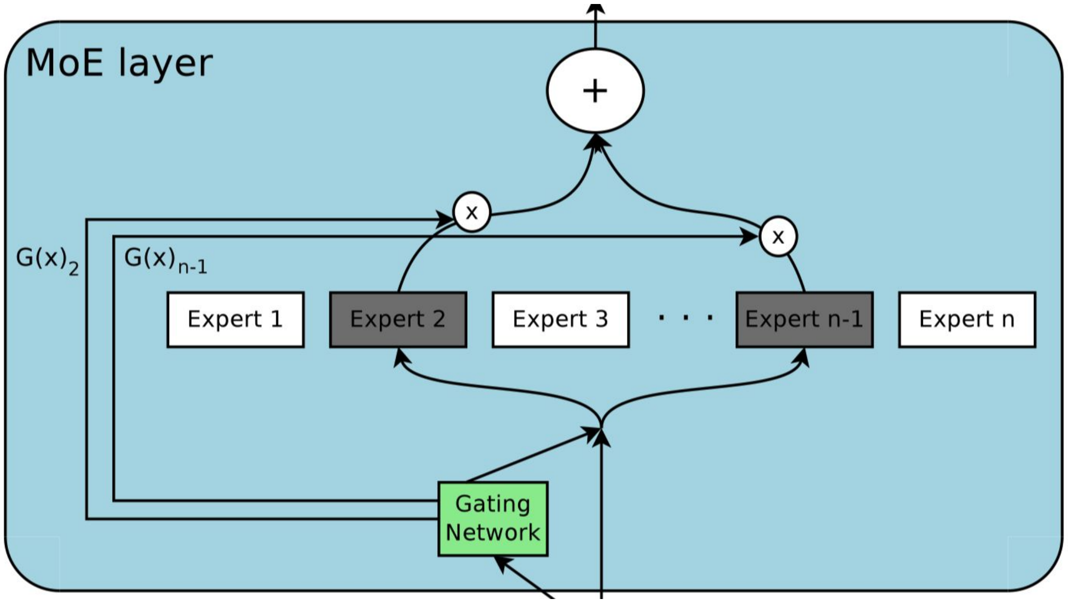
GShard

- Conditional computation
  - Activate sub-network on per-input basis
  - Sub-linear cost when scaling
  - Sparsely Gated Mixture-of-Experts (MoE)

- GShard Annotation API
  - Separate model description from partitioning implementation and optimization
  - Compiler extension in XLA for automatic parallelization
  - Special split/partition annotation for tensor
  - Special operator such as Einsum
  - Iterative data-flow analysis to infer sharding for the rest of the tensors

GShard





Carnegie Mellon University

MoE

- Load balancing
  - Prevent routing to a small number of experts
  - Expert Capacity
  - Overflowed token via residual connection

- Local dispatching for parallel gating
  - Partition batch into local groups
  - Independently in parallel
  - Fractional Expert Capacity
  - Speed up gating function by number of groups

- Auxiliary loss
  - Enforce the load balancing

- Random routing
  - Randomly ignore 2nd expert to conserve overall expert capacity

Gating Function

Algorithm 1: Group-level top-2 gating with auxiliary loss

Data: x_S, a group of tokens of size S
Data: C, Expert capacity allocated to this group
Result: $\mathcal{G}_{S,E}$, group combine weights
Result: ℓ_{aux}, group auxiliary loss

for $e \leftarrow 1$ **to** E **do**
- $c_e \leftarrow 0$ ▷ gating decisions per expert
- $g_{S,e} \leftarrow softmax(wg \cdot x_S)$ ▷ gates per token per expert, wg are trainable weights
- $m_e \leftarrow \frac{1}{S} \sum_{s=1}^{S} g_{s,e}$ ▷ mean gates per expert

end

Gating probability for each expert

for $s \leftarrow 1$ **to** S **do**
- $g1, e1, g2, e2 = top\_2(\{g_{s,e} | e = 1 \cdots E\})$ ▷ top-2 gates and expert indices
- $g1 \leftarrow g1/(g1 + g2)$ ▷ normalized $g1$
- $c \leftarrow c_{e1}$ ▷ position in $e1$ expert buffer
- **if** $c_{e1} < C$ **then**
  - $\mathcal{G}_{s,e1} \leftarrow g1$ ▷ $e1$ expert combine weight for x_s
- **end**
- $c_{e1} \leftarrow c + 1$ ▷ incrementing $e1$ expert decisions count

end

Route to 1st expert

$\ell_{aux} = \frac{1}{E} \sum_{e=1}^{E} \frac{c_e}{S} \cdot m_e$

Auxiliary loss

for $s \leftarrow 1$ **to** S **do**
- $g1, e1, g2, e2 = top\_2(\{g_{s,e} | e = 1 \cdots E\})$ ▷ top-2 gates and expert indices
- $g2 \leftarrow g2/(g1 + g2)$ ▷ normalized $g2$
- $rnd \leftarrow uniform(0, 1)$ ▷ dispatch to second-best expert with probability $\propto 2 \cdot g2$
- $c \leftarrow c_{e2}$ ▷ position in $e2$ expert buffer
- **if** $c < C \wedge 2 \cdot g2 > rnd$ **then**
  - $\mathcal{G}_{s,e2} \leftarrow g2$ ▷ $e2$ expert combine weight for x_s
- **end**
- $c_{e2} \leftarrow c + 1$

end

Randomly route to 2nd expert

Carnegie Mellon University

GShard Annotation

Algorithm 2: Forward pass of the Positions-wise MoE layer. The underscored letter (e.g., G and E) indicates the dimension along which a tensor will be partitioned.

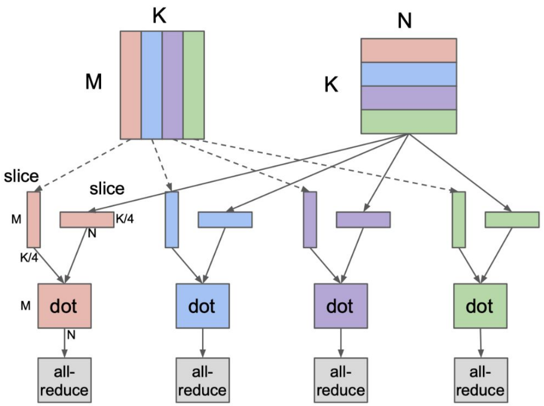
```
gates = softmax(einsum("GSM,ME->GSE", inputs, wg))
combine_weights, dispatch_mask = Top2Gating(gates)
dispatched_inputs = einsum("GSEC,GSM->EGCM", dispatch_mask, inputs)
h = einsum("EGCM,EMH->EGCH", dispatched_inputs, wi)
h = relu(h)
expert_outputs = einsum("EGCH,EHM->GECM", h, wo)
outputs = einsum("GSEC,GECM->GSM", combine_weights, expert_outputs)
```

```
  # Partition inputs along the first (group G) dim across D devices.
+ inputs = split(inputs, 0, D)
  # Replicate the gating weights across all devices
+ wg = replicate(wg)
  gates = softmax(einsum("GSM,ME->GSE", inputs, wg))
  combine_weights, dispatch_mask = Top2Gating(gates)
  dispatched_inputs = einsum("GSEC,GSM->EGCM", dispatch_mask, inputs)
  # Partition dispatched inputs along expert (E) dim.
+ dispatched_inputs = split(dispatched_inputs, 0, D)
  h = einsum("EGCM,EMH->EGCH", dispatched_inputs, wi)
```

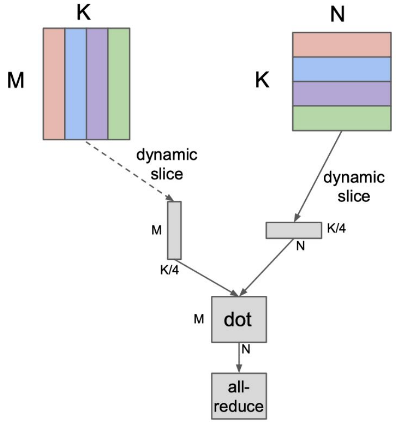
Carnegie Mellon University

XLA SPMD Partitioner For GShard

- Single Program Multiple Data
  - Transforms computation graph into single program to parallel execution
  - Constant compilation time regardless of the number of partitions



(a) MPMD Partition



(b) SPMD Partition

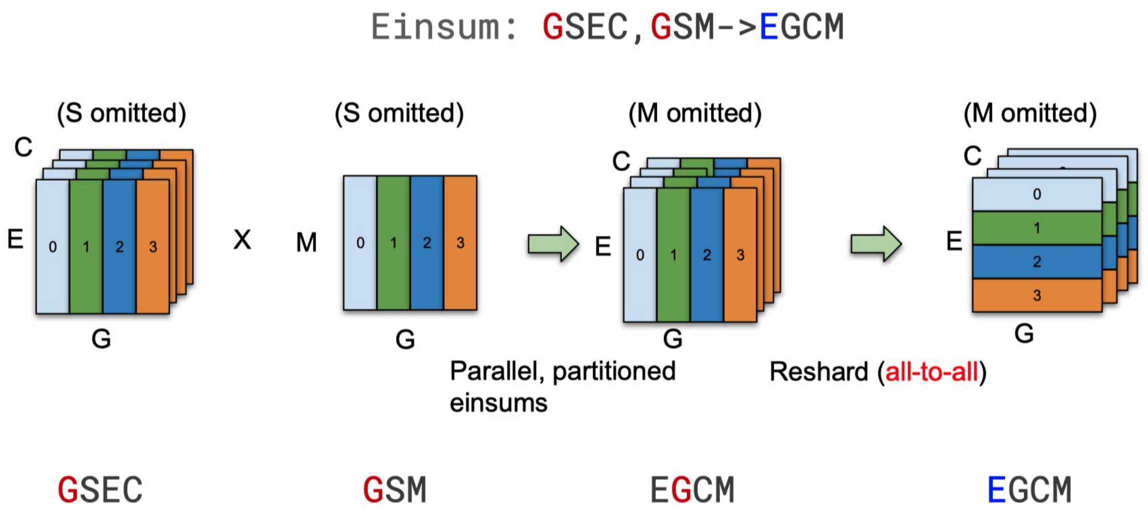
Carnegie Mellon University

Einsum

- Einstein Summation Notation
  - Specify dimensions for input and output tensor
  - “ij, jk -> ik”
- Resharding
  - Repartition in batch dimension
- Accumulating partial results
  - Partitioned along contracting dimensions
  - AllReduce
- Slicing in a loop
  - Limit size of tensor
  - Non-contracting dimensions

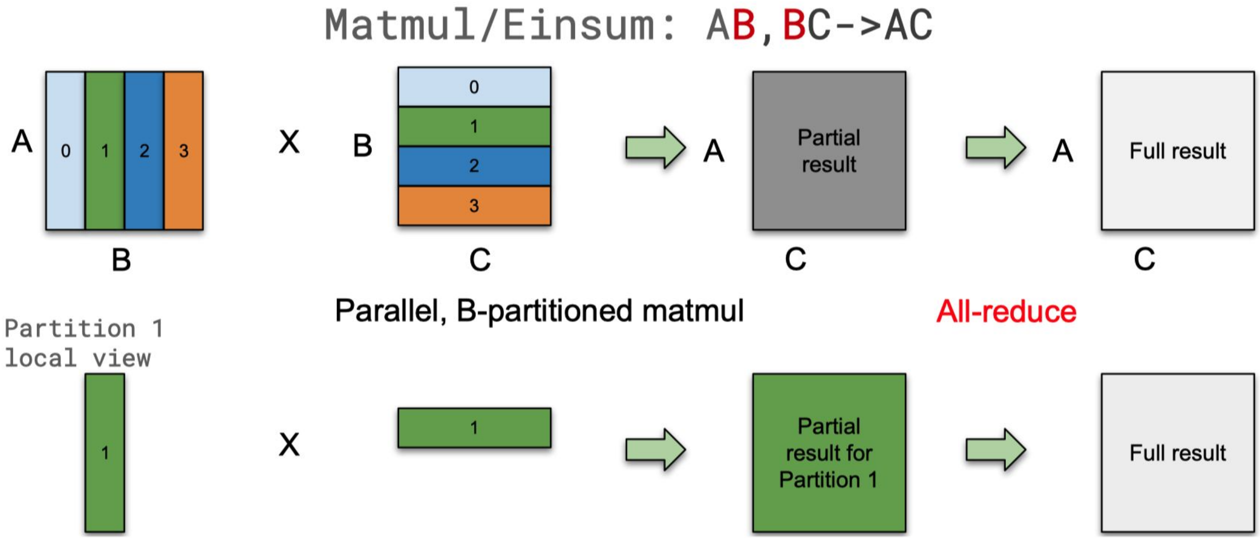
Einsum

Resharding



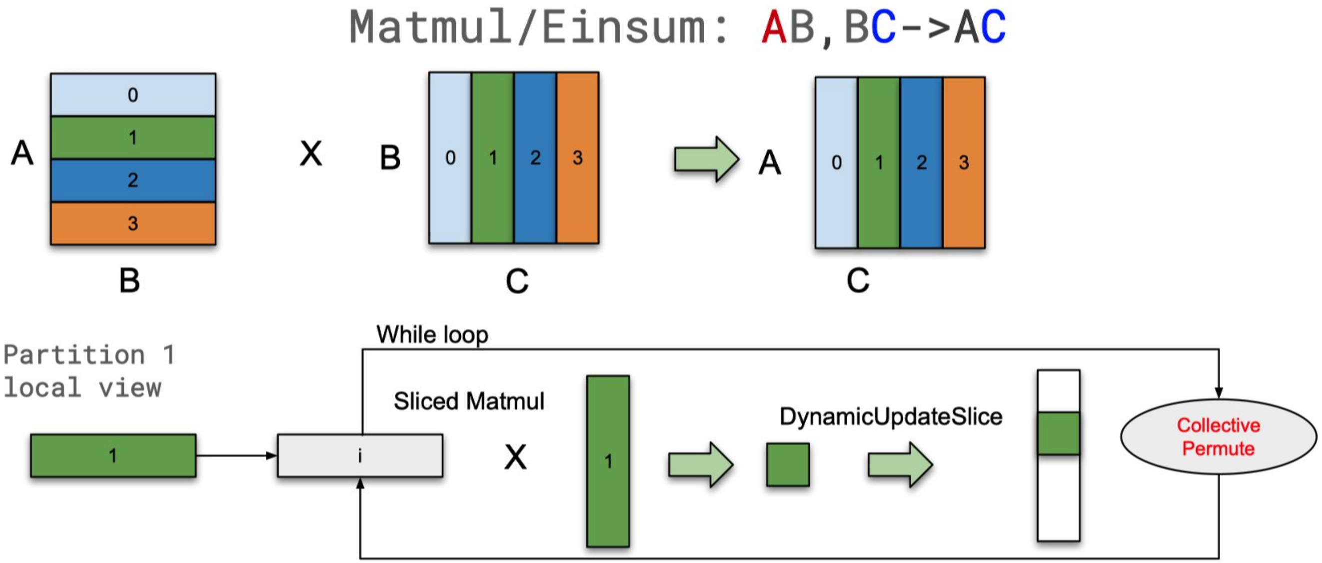
Einsum

Accumulating partial results



Einsum

Slicing in a loop

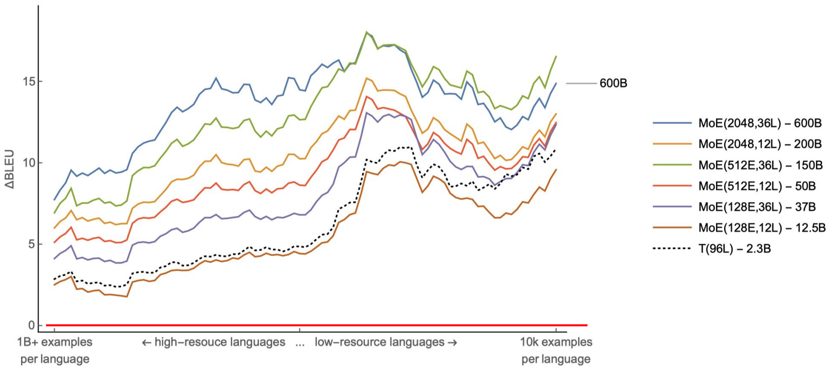


Massively Multilingual, Massive Machine Translation

- Single neural network translating multiple language pairs simultaneously
  - Between more than hundred languages
  - One trillion T tokens

- Positive transfer
  - Transfer expert knowledge
  - Sharing sub-networks
  - Benefit low-resourced languages

Results

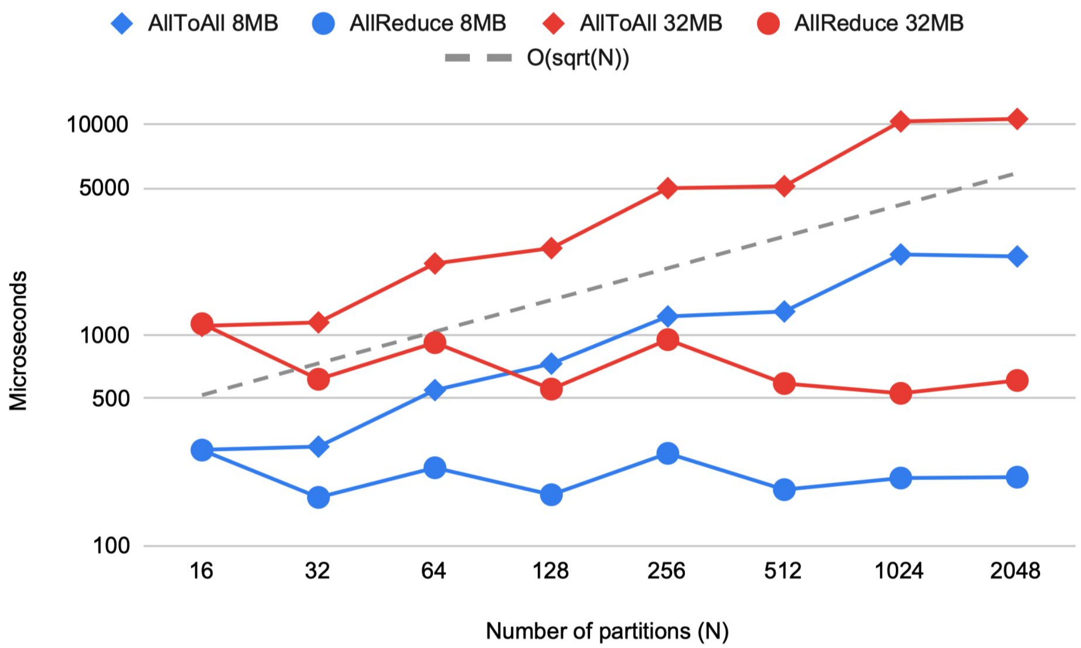


Model	Cores	Steps / sec.	Batch sz (Tokens)	TPU core years	Training days	BLEU avg.	Billion tokens to cross-entropy of 0.7	0.6	0.5
MoE(2048E, 36L)	2048	0.72	4M	22.4	**4.0**	**44.3**	82	175	542
MoE(2048E, 12L)	2048	2.15	4M	7.5	1.4	41.3	176	484	1780
MoE(512E, 36L)	512	1.05	1M	15.5	11.0	43.7	66	170	567
MoE(512E, 12L)	512	3.28	1M	4.9	3.5	40.0	141	486	-
MoE(128E, 36L)	128	0.67	1M	6.1	17.3	39.0	321	1074	-
MoE(128E, 12L)	128	2.16	1M	1.9	5.4	36.7	995	-	-
T(96L)	2048	-	4M	∼235.5	∼42	36.9	-	-	-
Bilingual Baseline	-	-		∼ 29	-	30.8	-	-	-

Table 1: Performance of MoE models with different number of experts and layers.

Carnegie Mellon University

Results



Carnegie Mellon University

Results

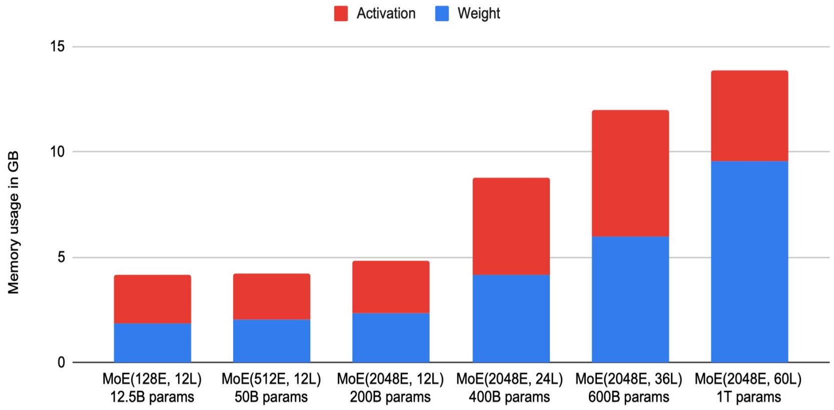
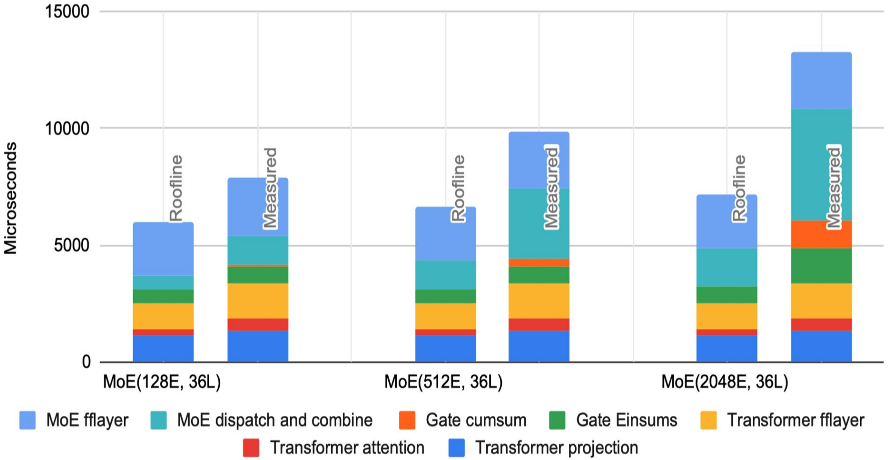


Figure 7: Per-device memory consumption in gigabytes.



Summary

- Conditional computation
- Sub-linear scaling
- Sparsely Gated Mixture-of-Experts (MoE)
- GShard Annotation API
- Compiler extension in XLA with constant SPMD compile time
- Single neural network translating multiple language
- Best result with 2048 MoE, 36 layers, trained for 4 days

Discussion

1. Why does the average BLEU gain for MoE(512E, 36L) exceed ones with higher capacity, but shallower MoE(2048E, 12L) in Table 1?

2. Given that there is diminishing return as we keep increasing the number of experts, but increasing layers adds more training time and not sub-linear cost, what would you do to improve the best model MoE(2048E, 36L) in the paper?

3. What sharding operator other than Einsum would you implement if you want to add more operations to GShard?