UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

EFFICIENTLY SCALING TRANSFORMER INFERENCE

Akul Gupta & Nachuan Wang

Introduction & Motivation

Transforming NLP with LLMs

- Breakthrough models like GPT-3 (100B+ parameters) and PaLM (500B+ parameters) have set new benchmarks in NLP
- **Why Scale?** Larger models lead to better generalization, outperforming smaller models across tasks.

Challenges in Efficient Inference

Memory Footprint

- **Large Models**: High memory usage from trained parameters and transient states during decoding.
- **KV Cache**: Attention key and value tensors must be stored for the duration of decoding
- **Memory Distribution**: Model parameters exceed the memory capacity of a single accelerator chip

Latency Constraints

- **Lower Parallelizability**: Generative inference is less parallelizable, making it difficult to meet tight latency targets.
- **Memory Bandwidth**: High memory traffic between HBM and compute cores due to large memory footprint

Quadratic Scaling of Attention Mechanism

- **Inference Cost**: Attention mechanism costs increase quadratically with input sequence length

Focus of This Paper: How can we achieve scalable, low-latency inference while meeting the unique demands of different applications?

- Present principles to optimize model partitioning strategies.
- Explore trade-offs between latency and throughput to support different production settings

Related Works

Existing Work: Parallelism & Inference Efficiency

1. Parallelism Approaches

- **NeMo Megatron (Korthikanti et al., 2022), GSPMD (Xu et al., 2021), Alpa (Zheng et al., 2022):**
  - Focus on efficient model partitioning for training large models.
  - Use tensor and pipeline parallelism with memory optimizations to improve performance.
- **FasterTransformer:**
  - Provides a benchmark for multi-GPU multi-node inference across models like Megatron–Turing NLG 530B.
  - Combines parallelism strategies and memory optimizations for faster inference.
- **DeepSpeed Inference (Aminabadi et al., 2022):**
  - ZeRO offload enables efficient use of CPU and NVMe memory alongside GPUs, especially beneficial for larger batch sizes.
- **EffectiveTransformer:**
  - Optimizes batch processing by packing sequences to minimize padding.

- This paper builds upon, **empirically-backed partitioning strategies** for efficient scaling of large models based on model size, context length, and chip count.

2. ML Inference Efficiency

- Several approaches focus on improving inference by optimizing **model architecture** and compression techniques:
  - **Efficient Attention Layers:** (Roy et al., 2020; Choromanski et al., 2020; Kitaev et al., 2020)
  - **Distillation & Pruning:** (Sanh et al., 2019; Li et al., 2020)
  - **Quantization:** (Dettmers et al., 2022; Zafrir et al., 2019)

- This paper incorporates prior work on **model quantization** to enhance inference speedups and could integrate further with **compression methods** like pruning.

Background

Inference Cost Tradeoffs & Breakdown

- Scaling up model sizes can unlock new capabilities but comes with fundamental tradeoffs.
- Key Metrics to measure inference cost:
    - **Latency**: Total time for an inference operation (prefill + decode)
        - **Prefill**: Time to process input tokens at the start.
        - **Decode**: Time to generate output tokens (can be per token, “per step”).
    - **Throughput**: Tokens processed/generated per second
    - **Model FLOPS (Floating Point Operations Per Second) Utilization (MFU)**: Efficiency of hardware use, ratio of observed throughput to theoretical peak FLOPS.

Compute Costs and Scaling Challenges

Compute Costs:

- **Matmul FLOPs**: Each forward pass for N-parameter models requires 2N matmul FLOPs per token.
- **Attention Mechanism**: Adds fewer FLOPs but increases memory costs due to unique KV cache for each sequence.

Scaling Challenges:

- **Low Latency Applications**: Require smaller batches and more partitioning across chips, but this leads to lower MFU and higher costs per token.
- **Long Attention Contexts**: Larger attention KV cache can cause bottlenecks, leading to longer inference times.
- **Offline Inference**: Focuses on maximizing throughput by increasing batch size for better MFU and lower cost per token.

Balancing latency, throughput, and memory is key for cost-effective scaling. Large models require careful management of memory and chip communication for optimal efficiency.

Background

Inference Setup & Notation

Consider a **Transformer model** with the following parameters:

- **n_params**: Number of parameters in the model
- **n_chips**: Number of accelerator chips used for inference
- **d_model (E)**: Model (or embedding) dimension
- **d_ff (F)**: Feedforward intermediate dimension
- **n_heads (H)**: Number of attention heads

Batch Setup and Token Handling

- Each batch of **B sequences** has:
  - **L_input**: Number of input tokens
  - **L_gen**: Number of output tokens generated

Inference Process

1. **Prefill Phase**:
   - Processes **B × L_input** tokens in parallel.
   - Single forward pass over all input tokens.
2. **Decode Phase (Autoregressive Generation)**:
   - **Sequential loop** of **L_gen** steps.
   - One forward pass per step generates one token for each of the **B** examples.

Key Differences in Performance

- **Prefill**: Can be parallelized for faster processing.
- **Decode**: Must run sequentially, leading to different performance characteristics.

Background

Partitioning

What is Model Partitioning?

- **Model partitioning** is the process of dividing a large neural network model (e.g., Transformer) across multiple devices (e.g., TPUs, GPUs).
- The goal is to distribute the model's **weights (parameters)** and **computations** across multiple chips to enable efficient **parallel processing**.

Key Strategies for Model Partitioning

Tensor Dimension Sharding:

- **Sharding** involves splitting a model's large tensors (e.g., weights) along one or more dimensions and distributing them across devices.
- Example: 1D & 2D Partitioning 2D Partitioning

Pipeline Parallelism:

- The model is **split layer-wise** so different layers of the model are distributed across multiple devices.
- Each device works on a subset of the layers, with outputs passed sequentially from one device to the next.

C. Data Parallelism vs. Model Parallelism:

- **Data Parallelism:** The same model copy is run on each device with different data batches.
- **Model Parallelism:** Different **portions of the model** are split and computed across multiple devices.

Communication in Partitioning

- **Inter-device communication** is required to synchronize the results of partial computations.

Partitioning Notation & Communication Collectives

- **System Overview:**
  - Partitioning based on **TPU v4 system** with a **3D torus topology** (X × Y × Z).
  - **Tensor Dimensions:**
    - **B** = Batch Size
    - **L** = Sequence Length
    - **E** = Embedding Dimension
    - **F** = MLP Feedforward Dimension
- **Partitioning Notation:**
  - **BLE_xyz**: Last dimension E of a tensor (shape BLE) is partitioned across **X × Y × Z** TPU axes.
    - Example: Tensor on each chip becomes **[B, L, E / (X × Y × Z)]**.
  - **Partialsum-x**: Indicates a tensor has been summed locally on the x-axis and requires further summing across chips.
- **Communication Collectives:**
  - **all-reduce(x)**: Sums a **partial sum** tensor across chips on the x-axis and broadcasts the result.
  - **reduce-scatter(x)**: Reduces the tensor across chips but outputs a **sharded** result instead of replicated.
  - **all-gather(x)**: Concatenates and broadcasts a sharded tensor back to all chips on the x-axis.
  - **all-to-all**: Shifts sharding from one tensor dimension to another using direct communication between chips.

TPU Interconnect Topology

1. What is TPU Interconnect Topology?

- TPU (Tensor Processing Unit) interconnect topology refers to the **physical structure** that defines how TPUs communicate with each other in a **multi-chip setup**.

2. 3D Torus Topology

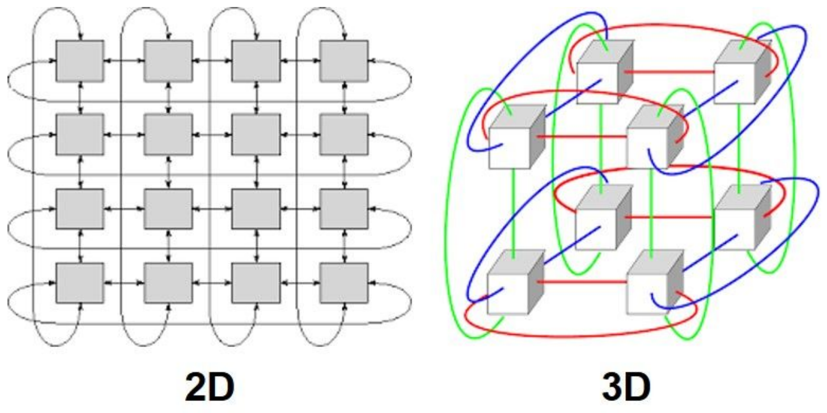
- **Google TPU v4** uses a **3D torus topology** to connect TPU chips in a **grid-like structure**:
  - Each chip is connected to its neighbors in **three dimensions (X, Y, Z)**.
  - This design allows for **low-latency communication** and **high bandwidth** between TPUs.

3.Advantages of 3D Torus:

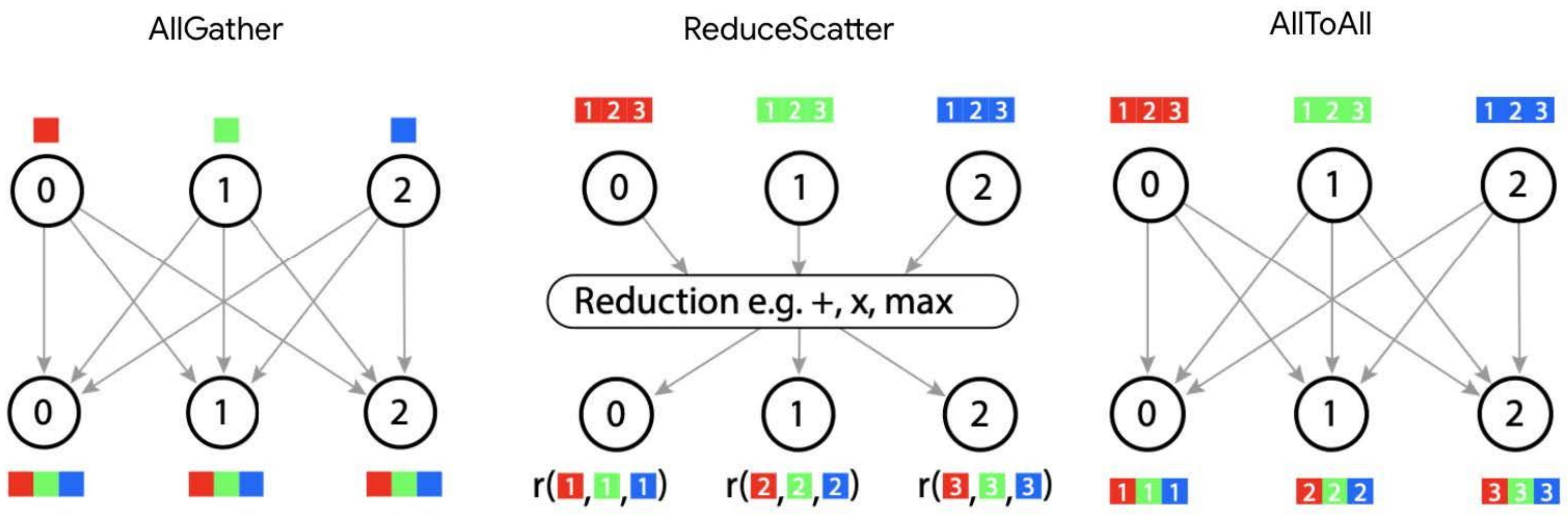
- **Scalability:** Supports a large number of TPUs interconnected in multiple directions.
- **Efficient Communication:** Reduces the communication bottleneck, enabling faster exchange of data
- **Fault Tolerance:** If a link fails, data can be rerouted using alternative paths.

4. How It Impacts Model Partitioning

- **Model partitioning** leverages the 3D torus to efficiently divide and distribute **model parameters** and computations
- Communication primitives like **all-reduce**, **reduce-scatter**, and **all-gather** are optimized for this topology, ensuring efficient data transfer and reducing overhead in large distributed models.



Collective Operations



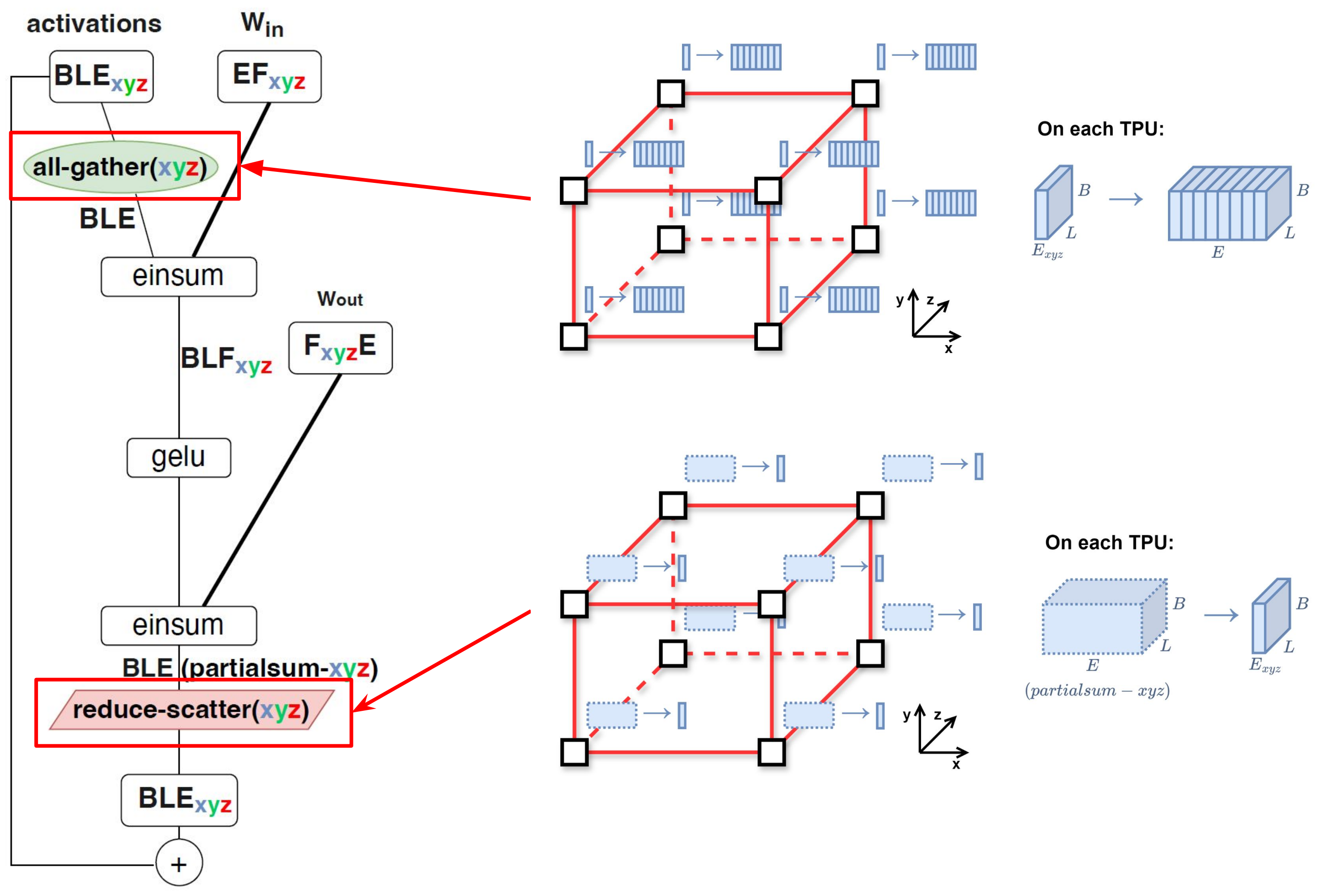
Number of chips: K
Size of full data: D
Network bandwidth between one pair of nodes: net_bw

Size of data chunk transferred between any pair of nodes: D/K
Time to transfer single data chunk: D/(K*net_bw)
Time to finish the collective operation: (D/(K*net_bw))*(K-1) ~= D/net_bw (assuming no link contention)

Implication: when K is large, T(collective) is independent of number of chips involved

Feedforward Layer Partition

1D weight-stationary



T(all-gather) = BLE / net_bw

+

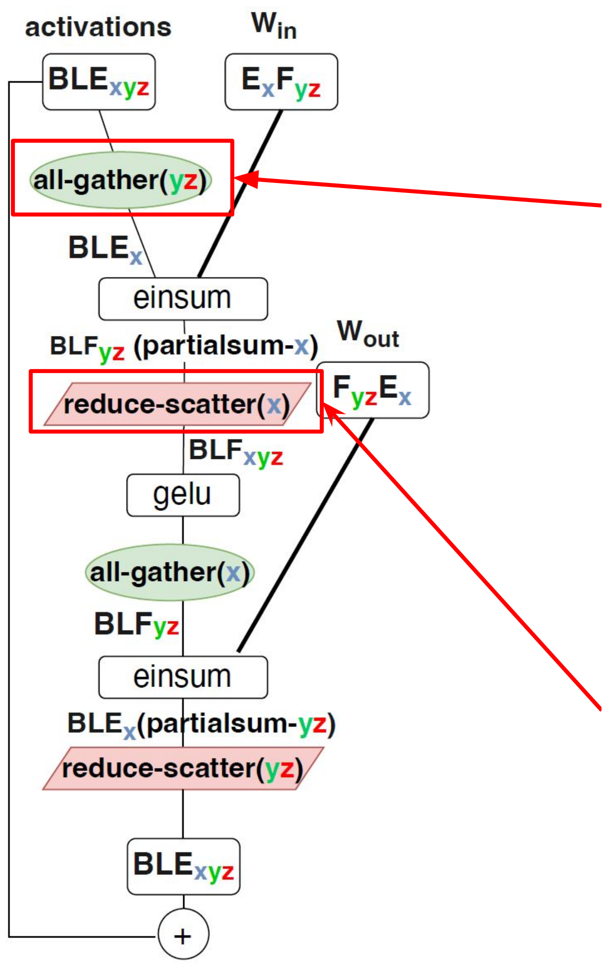
T(reduce-scatter) = BLE / net_bw

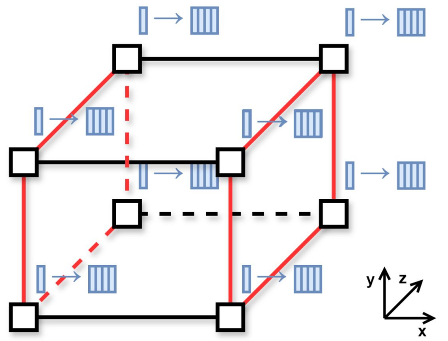
=

$$\boxed{\text{T(total) = 2BLE / net\_bw}}$$

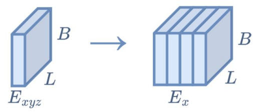
Feedforward Layer Partition

2D weight-stationary



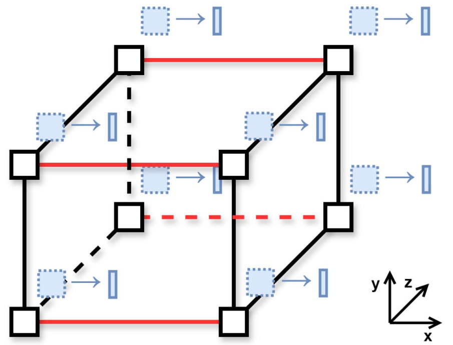


On each TPU:

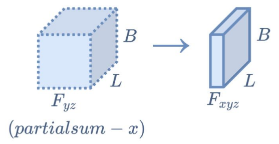


$$T(\text{all-gather}) = (BLE / X) / net\_bw$$

$$+$$



On each TPU:



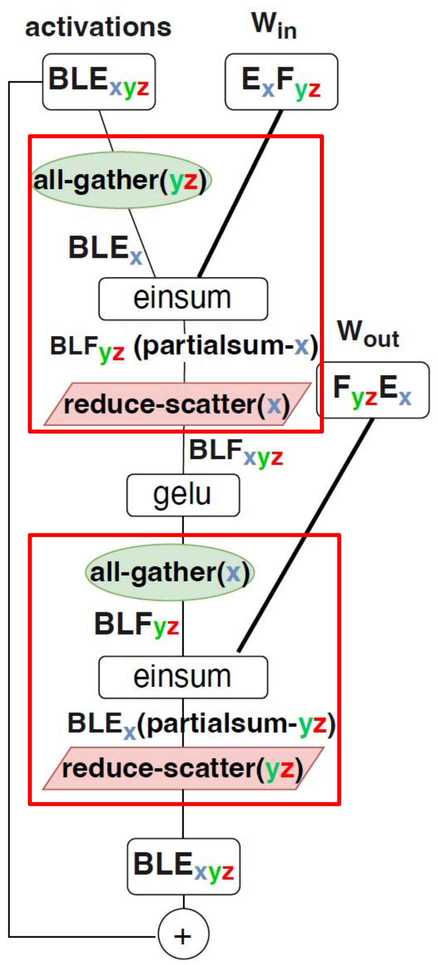
$$T(\text{reduce-scatter}) = (BLF / YZ) / net\_bw$$

$$=$$

$$\boxed{T = (BL / net\_bw) * (E/X + F/YZ)}$$

Feedforward Layer Partition

2D weight-stationary, cont.



T = (BL / net_bw) * (E/X + F/YZ)

+ =

T(total) = 2BL / net_bw* (E/X + F/YZ)

↓ YZ = n_chip/X

T = (BL / net_bw) * (E/X + F/YZ)

T(total) = 2BL / net_bw * (E / X + FX / n_chip)

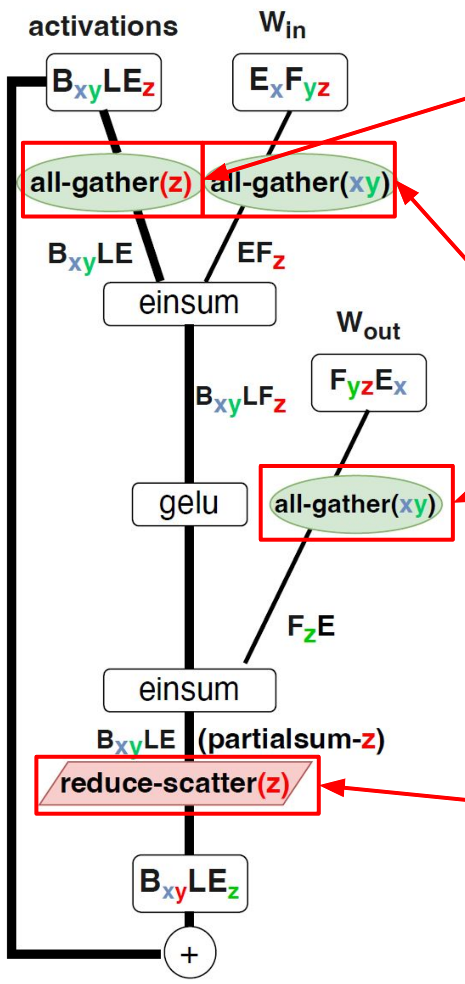
↓ When X = (E * n_chip / F)^½

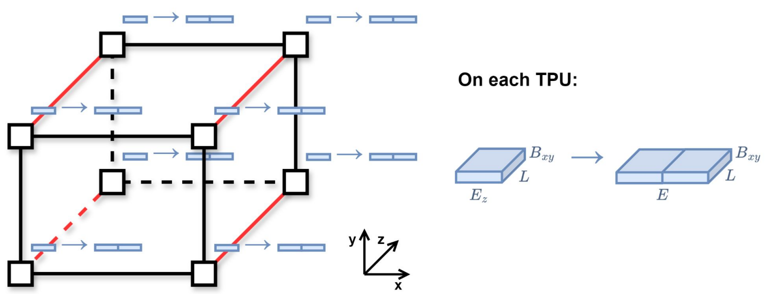
T(total, min) = (4BL / net_bw) * (EF / n_chip)^½

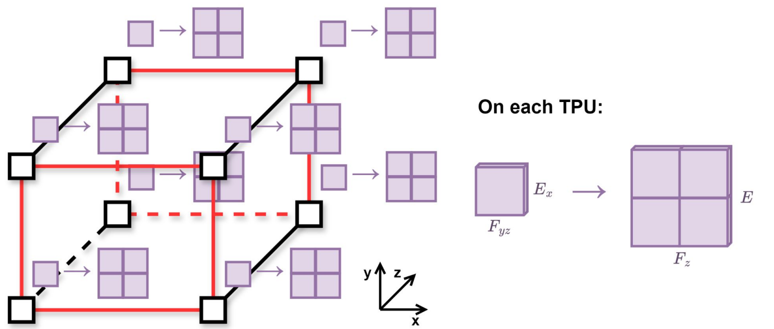
T scales as O(1 / n_chip^½)

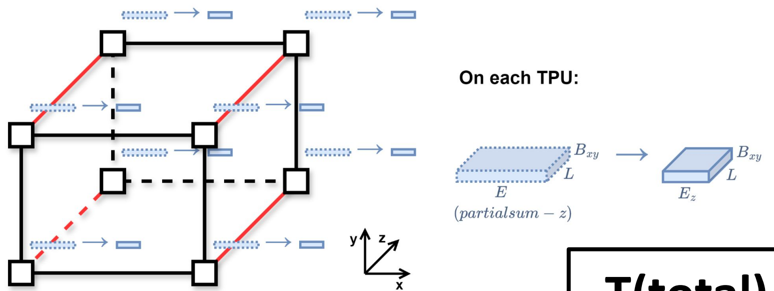
Feedforward Layer Partition

Weight-gathered









T(all-gather) = (BLE / XY) / net_bw

+

2 * T(all-gather) = (EF / Z) / net_bw

+

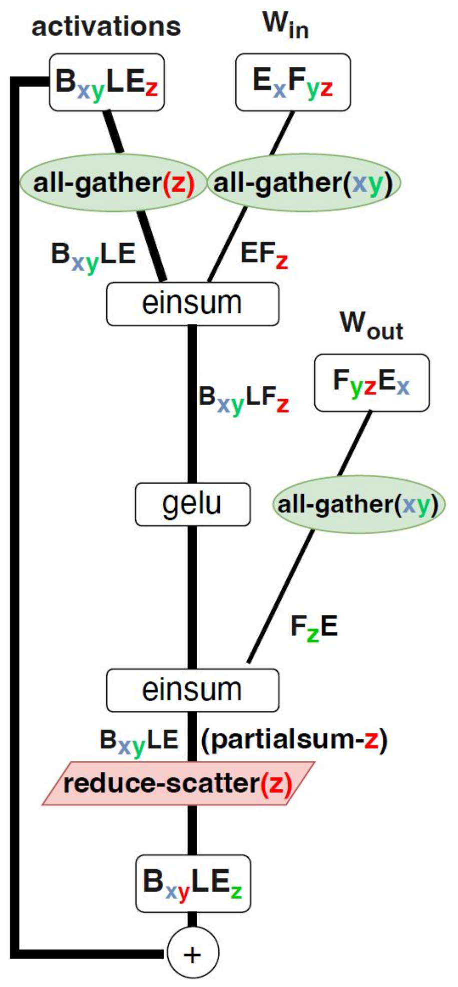
T(reduce-scatter) = (BLE / XY) / net_bw

=

T(total) = 2BLE / (net_bw * XY) + 2EF / (net_bw * Z)

Feedforward Layer Partition

Weight-gathered, cont.



T(total) = 2BLE / (net_bw * XY) + 2EF / (net_bw * Z)

$\downarrow$ XY = n_chip / Z

T(total) = (2E / net_bw) * (BLZ / n_chip + F / Z)

$\downarrow$ When Z = (F * n_chip / BL)^½

T(total, min) = (4E / net_bw) * (BLF / n_chip)^½ T scales as O((BL)^½)

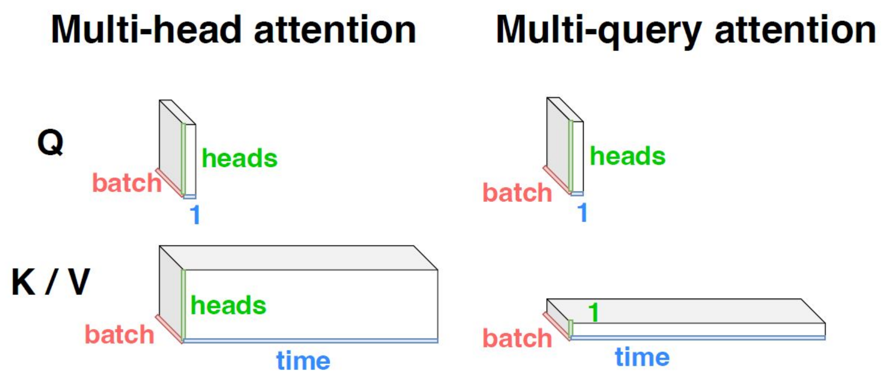
Comparing with 2D weight-stationary:

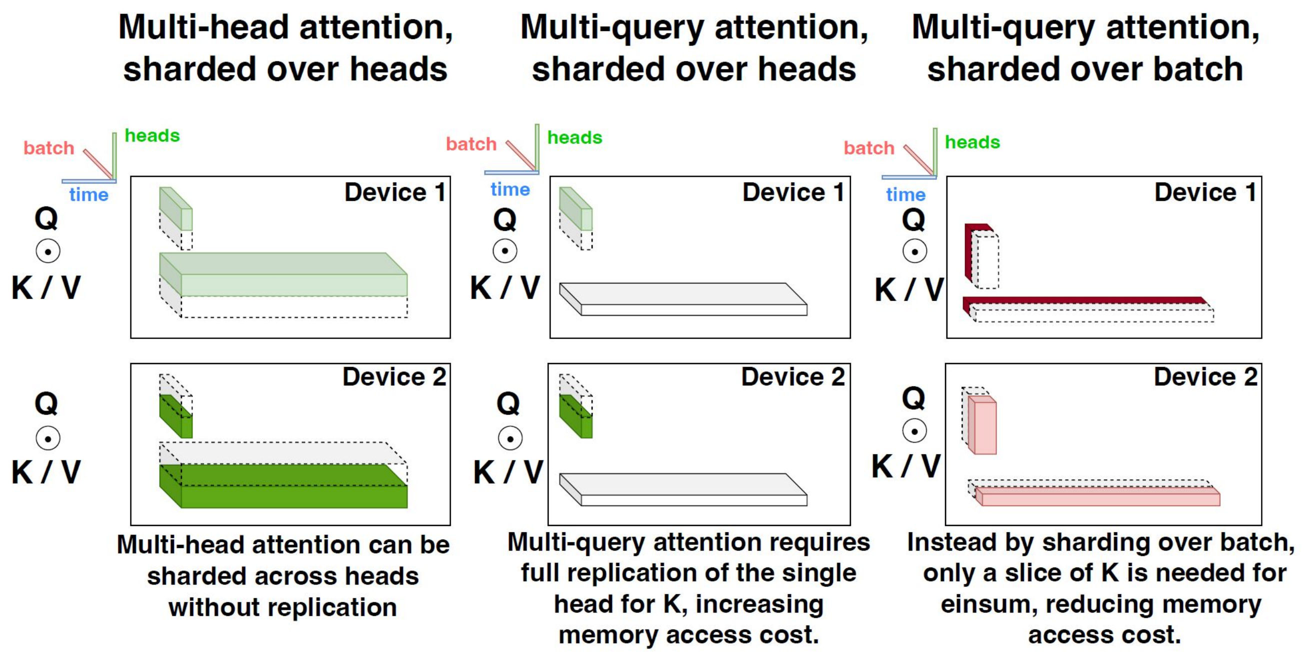
T(total, min) = (4BL / net_bw) * (EF / n_chip)^½ T scales as O(BL)

More efficient for large B and L

Attention Layer Partition

Multi-head vs. Multi-query





Multi-head attention, sharded over heads
Multi-query attention, sharded over heads
Multi-query attention, sharded over batch
batch
heads
time
Q
⊙
K / V
Device 1
Device 2

Multi-head attention can be sharded across heads without replication

Multi-query attention requires full replication of the single head for K, increasing memory access cost.

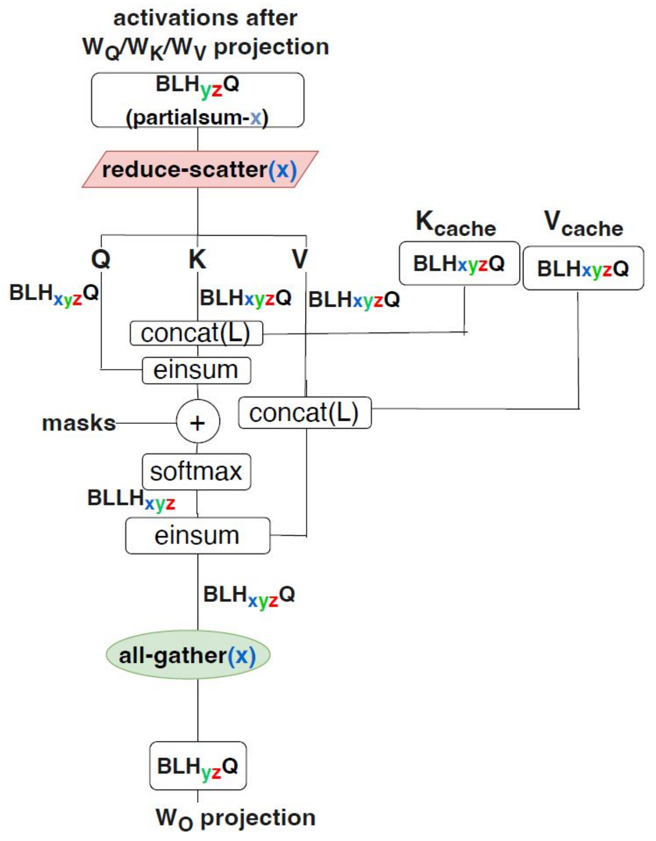
Instead by sharding over batch, only a slice of K is needed for einsum, reducing memory access cost.

Reduce memory footprint & load time of KV cache

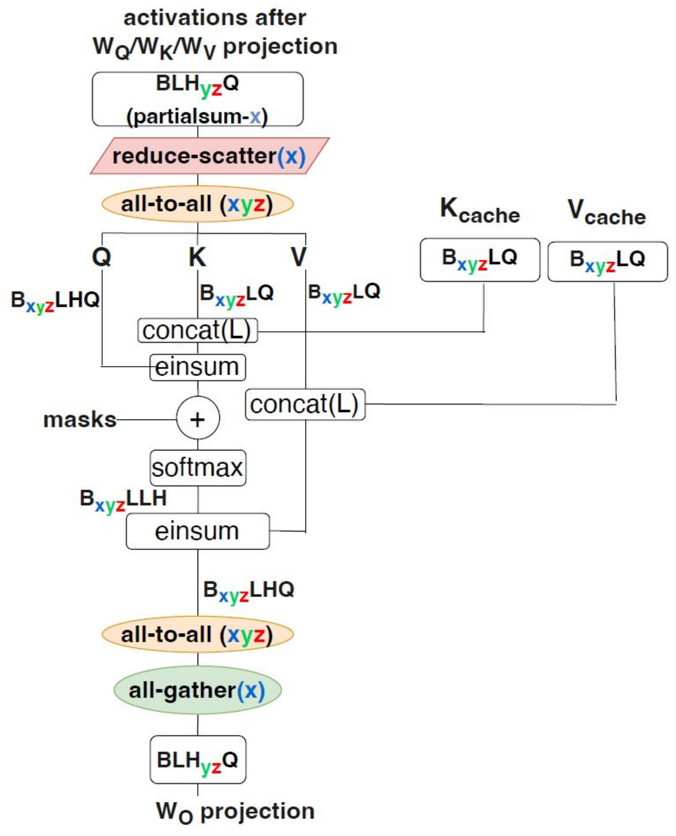
❌ ✅

Attention Layer Partition

Multi-head vs. Multi-query, cont.



Multi-head attention



Multi-query attention

Time spent in additional all-to-all operations:

$$T = 2(H+1)BQL\_new / (net\_bw * XYZ)$$

Time saved in KV cache access:

$$T = 2(H-1)BQL\_cache / (mem\_bw * XYZ)$$

Profitable when L_cache >> L_new

Case Study on PaLM Models

Methodology

- **256 TPU v4 chips, each with**
  - **275 TFLOPS BF16 throughput**
  - **32 GiB HBM, 1200 GB/s**
  - **3D torus interconnect, 270 GB/s**
- **PaLM 540B**

Case Study on PaLM Models

Partitioning Feedforward Layer

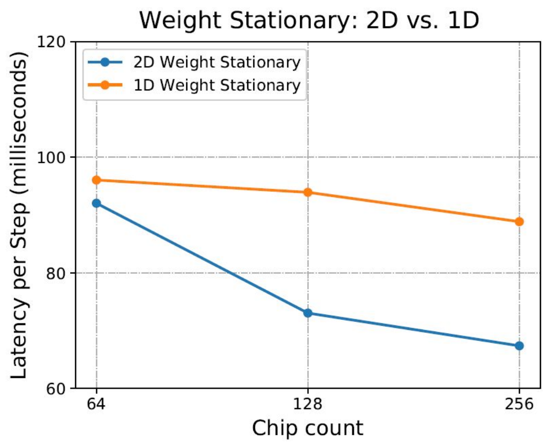


Figure 6: Latency per token doing text generation of PaLM 540B for 2D and 1D weight stationary layouts on 64 chips.

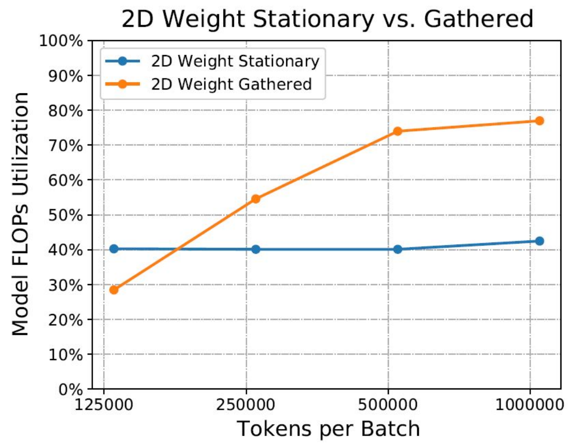


Figure 7: Model FLOPS utilization running prefill on PaLM 540B on 64 chips, with sequence length 2048. We report batch size measured in tokens: number of sequences multiplied by sequence length. As batch size (in tokens) grows, note that it is better to switch from the 2D weight stationary to the weight gathered approach to improve MFU.

Decode phase: 2D weight-stationary performs better as as chip count increases (T(comm) scales as O(1 / n_chip^½))

Prefill phase: weight-gathered can achieve higher MFU before communication becomes a bottleneck (T(comm) scales as O((BL)^½))

Case Study on PaLM Models

Partitioning Attention Layer

Model variant	d_{head}	Max context length	
		batch=128	batch=512
Multihead	128	1320	330
Baseline multiquery	256	660	165
Optimized multiquery	256	43,000	10,700

Table 1: Maximum context length supported for different attention variants of PaLM 540B on 64 chips. We reserve 30% of the total memory for KV cache. Optimized multiquery attention enables up to 32x larger context lengths.

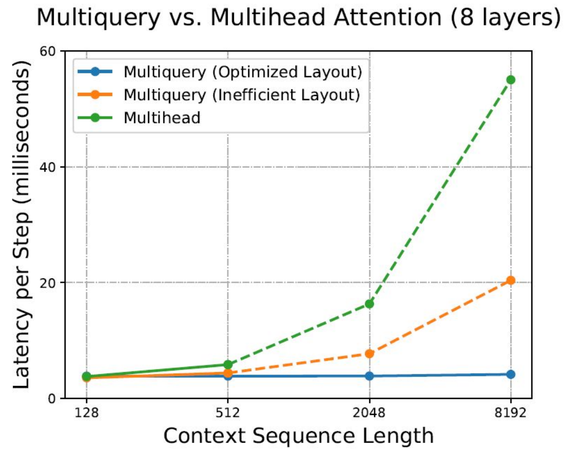


Figure 8: Latency per generated token vs. sequence length, for an 8-layer version of PaLM 540B on 64 chips with batch size 256. The dotted line represents that on the full 118-layer model and context lengths longer than 512, the KV cache will not fit in memory when using multihead attention or the baseline multiquery partitioning.

Optimized multiquery layout can fit up much larger context length by reducing memory footprint of KV cache

Much longer sequence generation with almost constant latency per-step

Case Study on PaLM Models

End-to-end Result

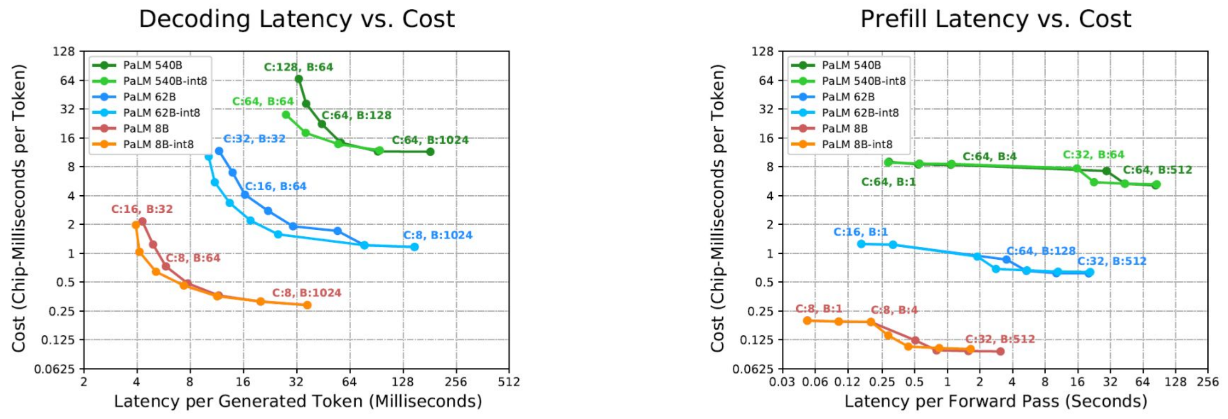


Figure 1: Cost vs. latency for PaLM models. We use a context length of 2048. Points in each line represent the Pareto frontier of efficiency versus latency. Chip count is C, batch size is B. Left: latency per token for generating 64 tokens, assuming the context has already been processed. Right: time to process 2048 input tokens; excludes the time to generate any output tokens. Tables 2 and 3 show details on a few specific scenarios from the Pareto frontier where the applications have low-latency or high-throughput requirements.

Decoding phase:

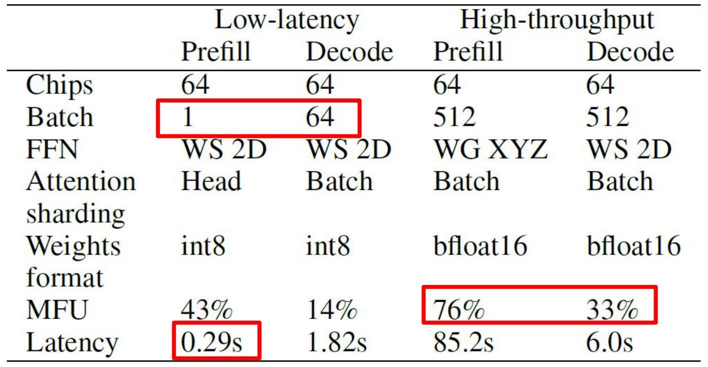
- **Distinct cost between int8/bf16 at low batch size**

Prefill phase:

- **Trade-off between cost and batch size is less severe**
- **Smaller cost than decoding phase due to its high MFU**

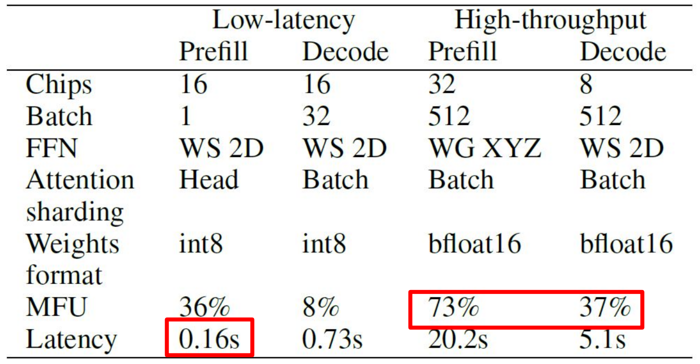
Case Study on PaLM Models

End-to-end Result, cont.



	Low-latency		High-throughput	
	Prefill	Decode	Prefill	Decode
Chips	64	64	64	64
Batch	1	64	512	512
FFN	WS 2D	WS 2D	WG XYZ	WS 2D
Attention sharding	Head	Batch	Batch	Batch
Weights format	int8	int8	bfloat16	bfloat16
MFU	43%	14%	76%	33%
Latency	0.29s	1.82s	85.2s	6.0s

Table 2: Example configurations for PaLM 540B, in the same setting as Figure 1. Prefill latency is for processing 2048 tokens; decode latency is for generating 64 tokens. Feedforward network (FFN) layouts are Weight Stationary 2D (WS 2D, Section 3.2.2) and Weight Gathered XYZ (WG XYZ, Section 3.2.3). Attention layouts are from Section 3.3.



	Low-latency		High-throughput	
	Prefill	Decode	Prefill	Decode
Chips	16	16	32	8
Batch	1	32	512	512
FFN	WS 2D	WS 2D	WG XYZ	WS 2D
Attention sharding	Head	Batch	Batch	Batch
Weights format	int8	int8	bfloat16	bfloat16
MFU	36%	8%	73%	37%
Latency	0.16s	0.73s	20.2s	5.1s

Table 3: Example configurations for PaLM 62B, in the same setting as Figure 1. Prefill latency is for processing 2048 tokens; decode latency is for generating 64 tokens. Feedforward network (FFN) layouts are Weight Stationary 2D (WS 2D, Section 3.2.2) and Weight Gathered XYZ (WG XYZ, Section 3.2.3). Attention layouts are from Section 3.3.

Lower-latency:

- **Combine 1-batch prefill with 64-batch decode to increase MFU**
- **Low-batch-size latency grow sub-linearly with model size at the Pareto frontier**

High-throughput:

- **MFUs are similar between the model sizes**

FasterTransformer Benchmarks

Different Hardware Configurations:

- FasterTransformer: 16–32 NVIDIA A100s (80 GiB HBM)
- Paper Setup: 64 Google TPU v4 chips (32 GiB HBM)
- Results normalized using *MFU* (Multichip FLOP Utilization) for fair comparison

Key Benchmark Results:

- Benchmarked models:
  - *Megatron 530B* (Smith et al., 2022)
  - *PaLM 540B* (with multiquery attention & parallel attention/feedforward)
- **Performance Highlights:**
  - *PaLM 540B*: Best absolute latency & up to 10% better MFU than Megatron
  - Parallel layers improve MFU, although advantages are offset by larger *dmodel* in Megatron

FasterTransformer Scalability:

- FasterTransformer:
  - 32-way tensor parallelism max = 33% MFU
  - 16-way tensor parallelism max = 46% MFU
  - Communication bottleneck observed at higher parallelism
- **Paper Implementation:**
  - Scales up to 64-way tensor parallelism with 44% MFU
  - Uses 2D weight-stationary partitioning strategy, demonstrating superior scalability on TPU v4

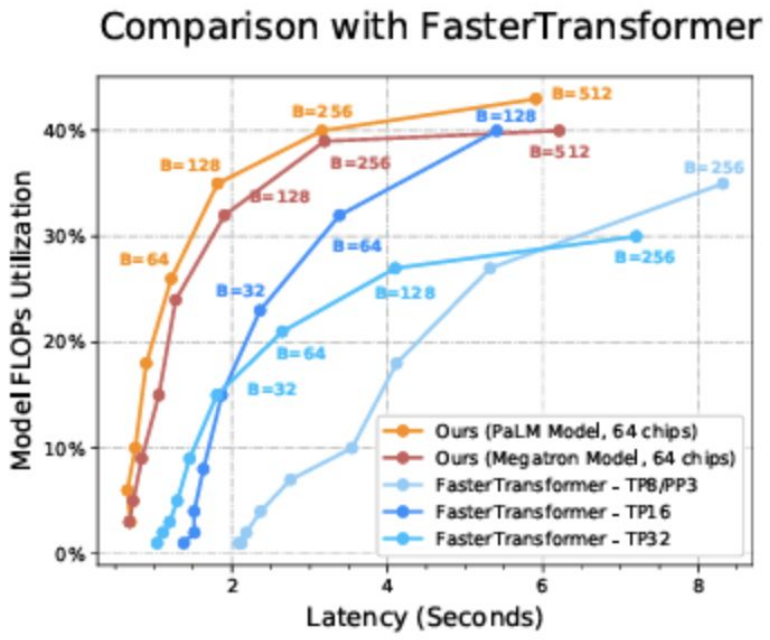


Figure 9: Model FLOPS utilization (MFU) versus total latency for running a 60-input-token, 20-output-token inference, at a range of batch sizes.

Our Thoughts

Strengths:

- **Comparative Evaluation:** Strong empirical evidence of improvements by comparing performance against established benchmarks like FasterTransformer and Megatron.
- **Scalability:** Multiquery attention allows the model to handle up to 32× larger context lengths, ideal for long-form content generation and document understanding.

Weaknesses:

- **Focus on Dense Models:** The paper focuses on dense Transformer models, with limited exploration of sparsity techniques (e.g., Mixture of Experts) that could reduce FLOPs and memory costs.
- **Limited Scalability Discussion:** While scaling to 64 TPU v4 chips is discussed, there’s limited exploration of potential bottlenecks beyond this number, such as communication overheads.

Future Directions:

- **Incorporating Sparsity:** Further exploration of sparsity techniques like Mixture of Experts (MoE) to reduce FLOPs and improve scalability in larger models.
- **Extending Quantization:** Investigate activation quantization to further reduce inference costs, particularly in high-throughput applications.

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Thank You

Q/A