CS 577: NATURAL LANGUAGE PROCESSING

Abulhair Saparov

Lecture 11: Efficiency II

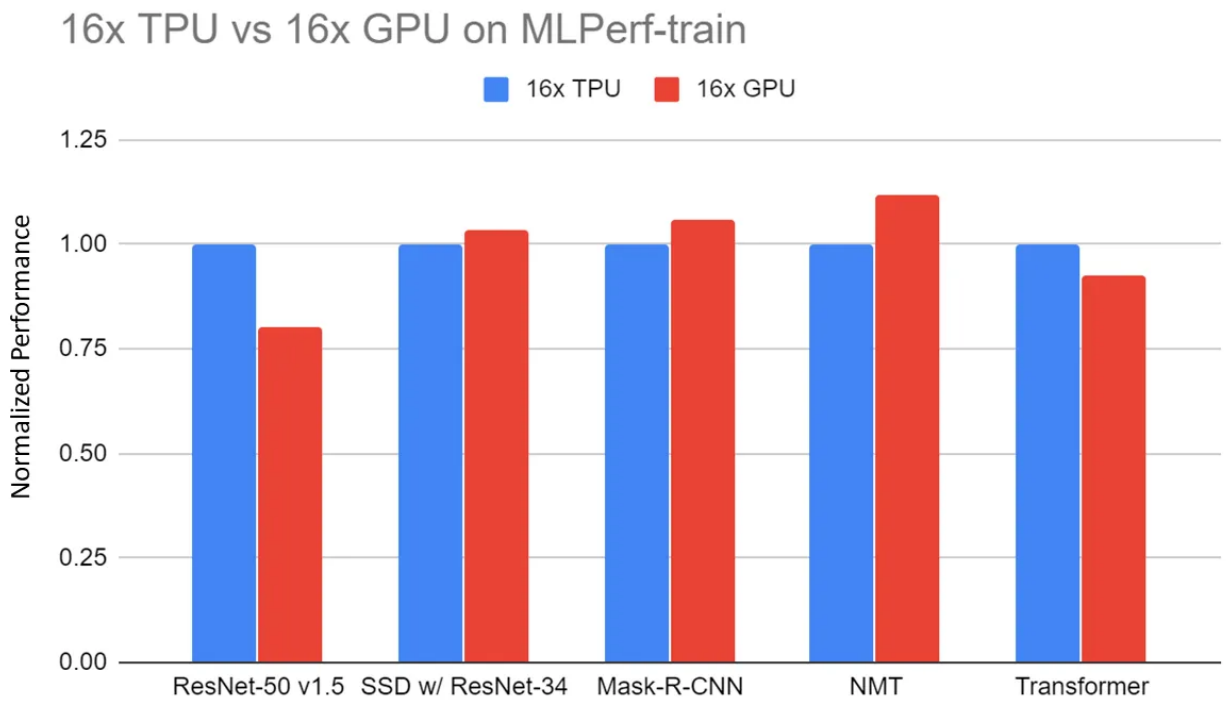
PREVIOUSLY: COST OF NLP MODELS

- We discussed how to compute the cost of running NLP models,
  - Both in terms of memory and time,
  - For training and inference.
- We talked about how we can make models run faster via parallelization.
  - And how to use hardware that is better suited for highly-parallel computation, such as GPUs.
  - GPUs have many cores compared to CPUs, even though each core is slower/more limited.

MORE SPECIALIZED HARDWARE?

- Google developed Tensor Processing Units (TPUs) specifically to accelerate common operations in deep learning, such as matrix multiplication.

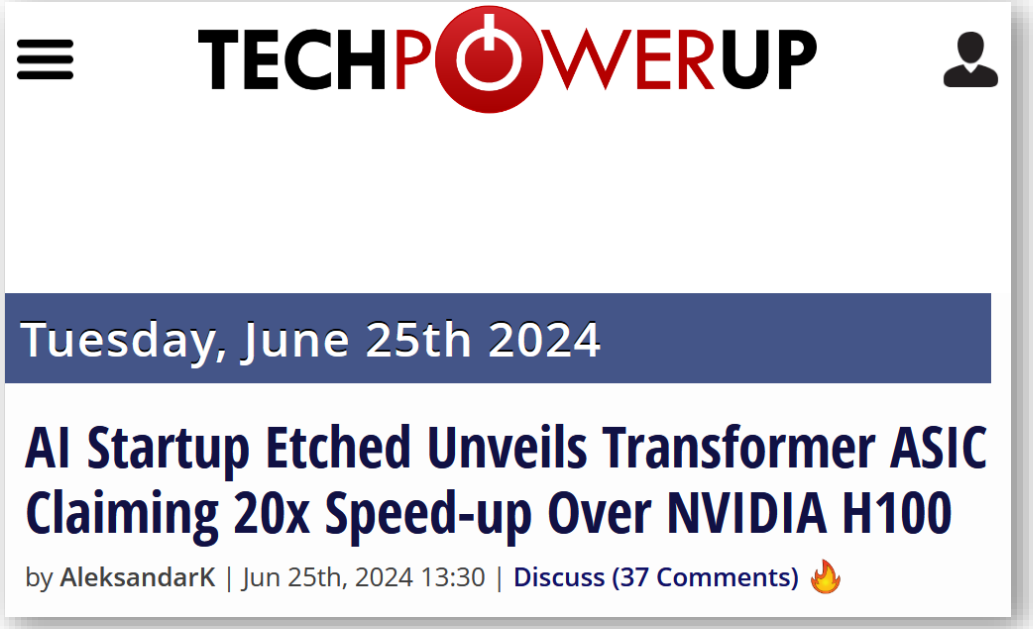
Performance is slightly better.



[Mahmoud Khairy, TPU vs GPU vs Cerebras vs Graphcore: A Fair Comparison between ML Hardware, 2020]

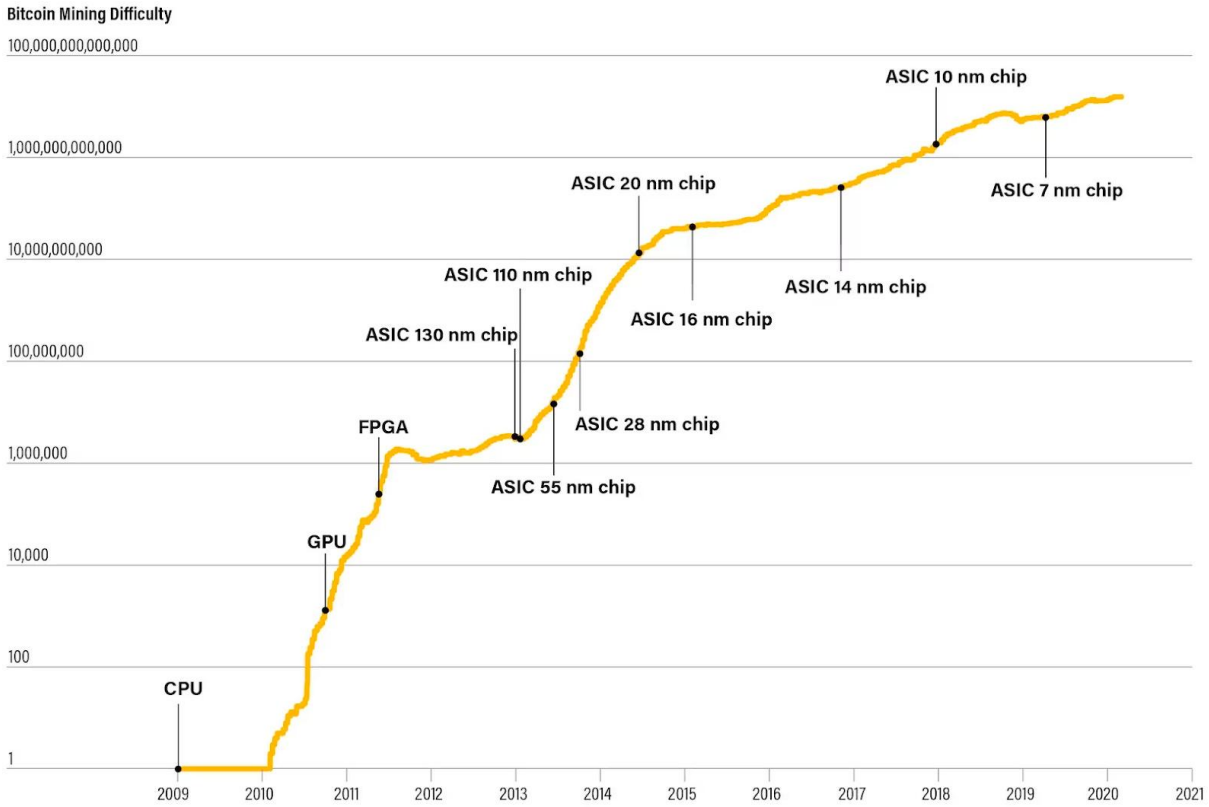
MORE SPECIALIZED HARDWARE?

- Many companies are currently developing application-specific integrated circuits (ASICs) which are specifically designed to accelerate transformer computations.



MORE SPECIALIZED HARDWARE

- A similar trajectory occurred with Bitcoin mining hardware.



[Austin Lyons, ASICs: Custom Chips to Improve Performance, 2024]

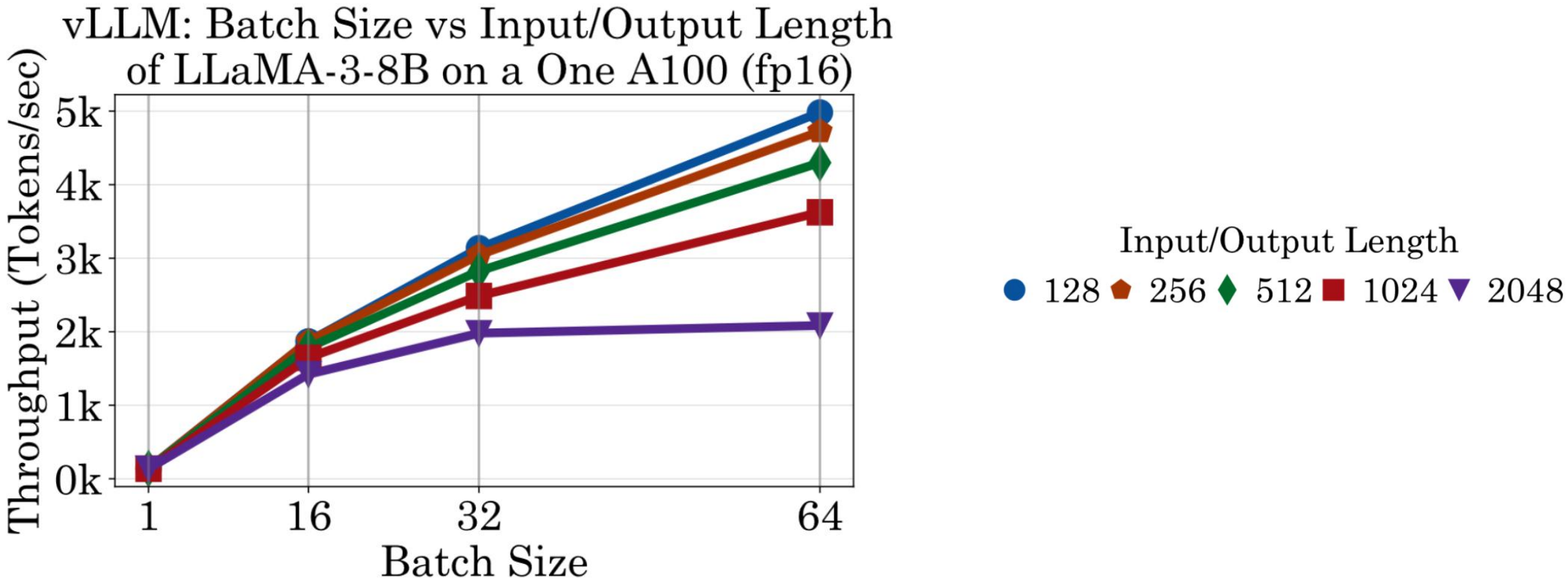
HOW TO MAKE TRANSFORMERS FASTER?

- NLP models can be very expensive, in terms of time and memory.
  - Especially as they are scaled up.
- Better hardware, such as GPUs, can really help to improve the performance of such models.
  - Transformers especially benefit from parallelization.
- Can we modify the transformer architecture to run faster?
  - Without sacrificing accuracy?
  - How small/fast can we make the model if we do tradeoff some accuracy?
  - Parameter-efficient fine-tuning
  - Model compression (quantization, distillation)
- What if the models/matrices can’t fit in the memory of one GPU?

BATCHING

- Batch size is an important hyperparameter during training.
- If the (mini-)batch size is too small, the gradient is too noisy,
  - And learning can be unstable.
- As we observe from scaling laws, larger models require larger batch sizes for optimal training.
- What about during inference?
  - Batching doesn’t make the model behave any differently,
  - But it can help to perform inference faster.

BATCHING

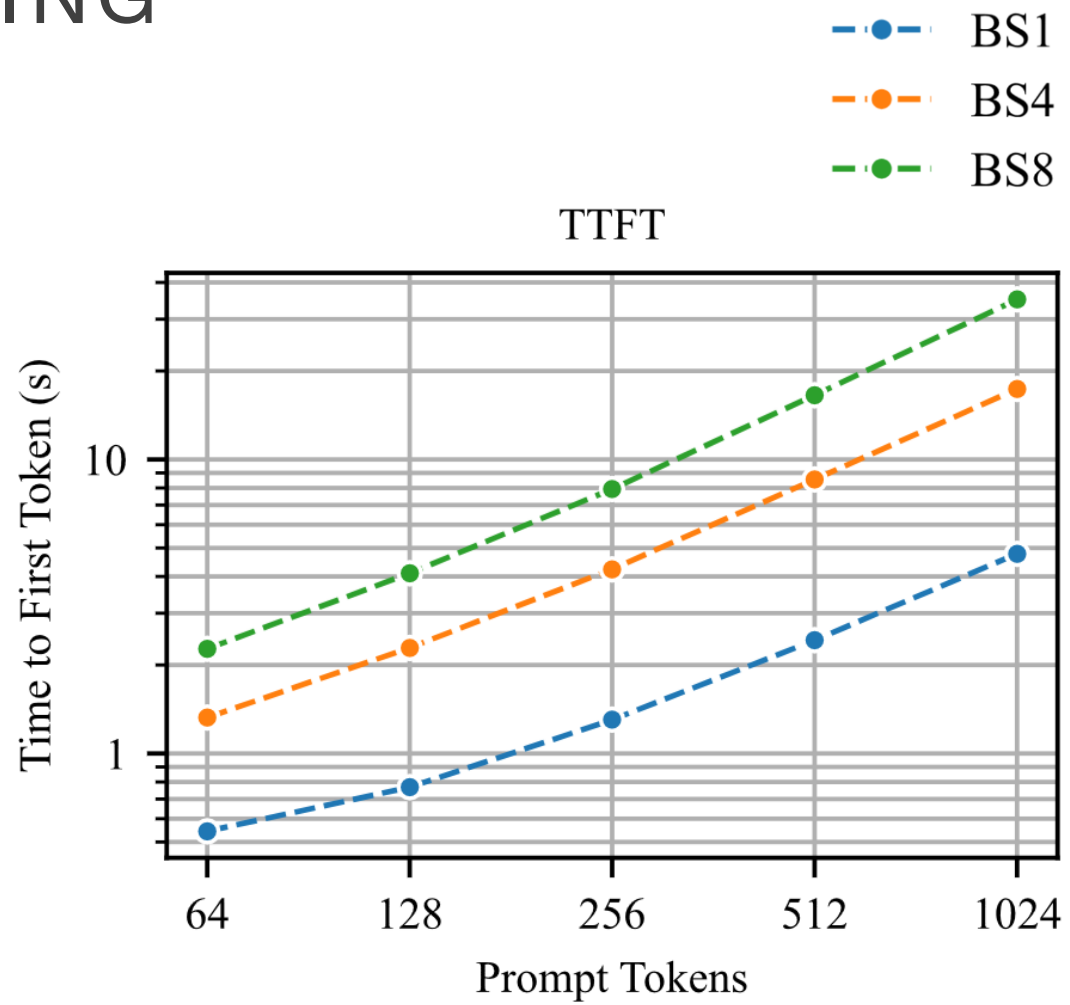


[Chitty-Venkata et al., 2024]

BATCHING

- But how does this work?
- There is a linear amount of additional input.

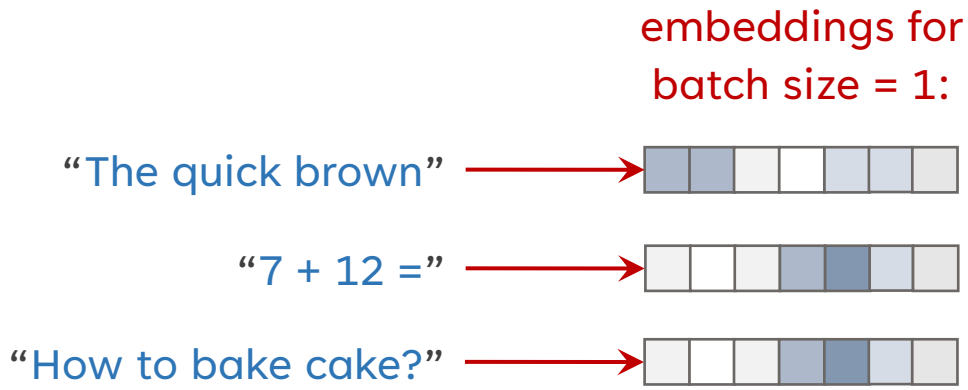
  (i.e., a linear amount of work to do)
- Shouldn't the total computation time also increase linearly?
- If we performed each forward pass one after the other, then yes.



[Horton et al., 2024]

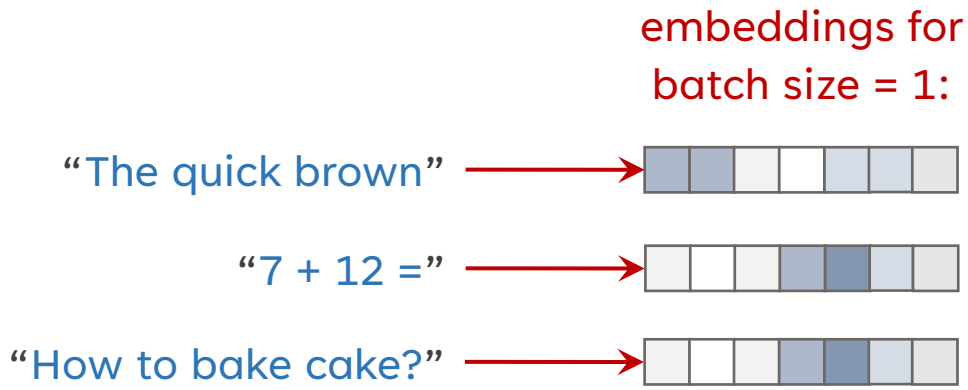
BATCHING

- Instead, we leverage the parallel processing ability of the hardware (GPU).
- Consider the forward pass of an RNN for some input.
  - The hidden state is a d-dimensional vector.
  - If we run separate forward passes for each of the three example inputs, we will have three different d-dimensional hidden state vectors at time t.



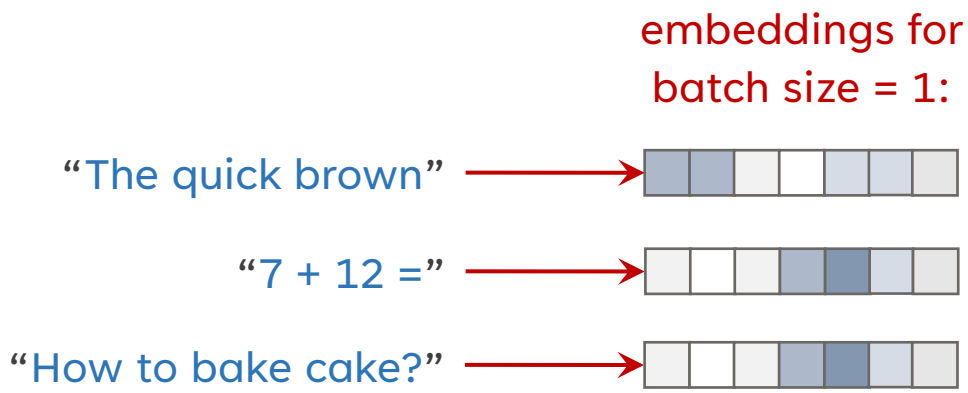
BATCHING

- Each of these hidden states are then processed at each step in the RNN's forward pass.
- Recall from last lecture: The most computationally-intensive part of this is the matrix multiplication in the linear layers.
- If we perform the forward passes separately, each matrix multiplication will compute the product of a $1 \times d$ vector and a $d \times d$ matrix.



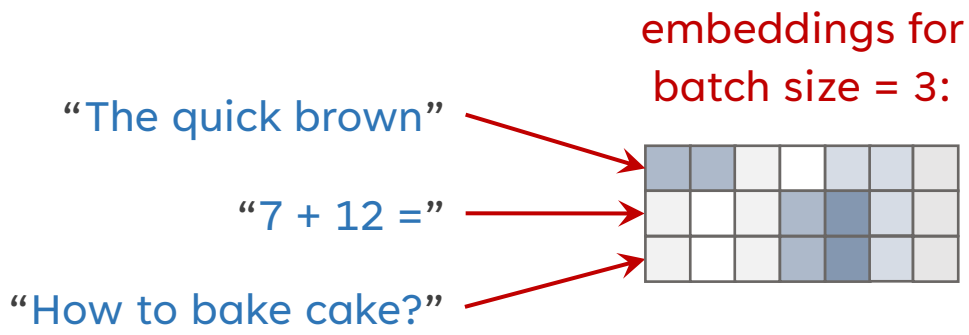
BATCHING

- So we need $2d^2$ FLOPs to compute the matrix product.
- If we have d threads, we can compute it in $2d$ time, using block matrix multiplication.
- But if we have even more threads, we won't see much more of a speedup.
- What there are more than d available threads? More than $2d^2$?



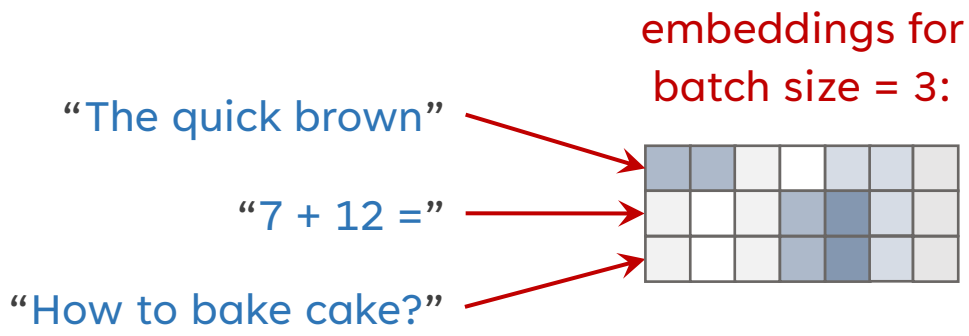
BATCHING

- If d is not too large, if batch size = 1, we are underutilizing parallel compute resources.
- Instead, we can concatenate the hidden state vectors across multiple forward passes into a single matrix with dimension $B \times d$, where B is the batch size.
- Then each linear layer in the RNN involves a product of a $B \times d$ matrix with a $d \times d$ matrix.



BATCHING

- The other operations in the RNN (e.g., softmax) are performed on each row of this matrix.
- In general, increasing batch size causes greater utilization of the hardware's "parallelization capacity."
- A more common measure of "parallelization capacity" is memory bandwidth.
  - How quickly can you write data to the GPU's high bandwidth memory?

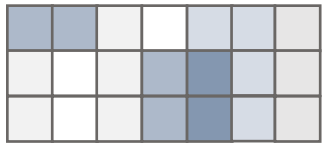


BATCHING

- We can easily apply batching to other models as well.
- For transformers, recall that the embedding is a matrix with dimension $n \times d_{model}$.
- When batch size > 1, the embedding becomes a tensor with dimension $B \times n \times d_{model}$.
- (conventionally, the batch is typically the first index)

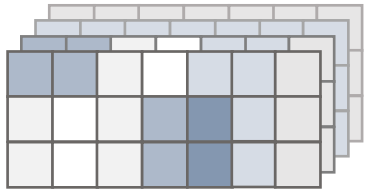
vector with size 7

(order-1 tensor)



matrix with size 3×7

(order-2 tensor)

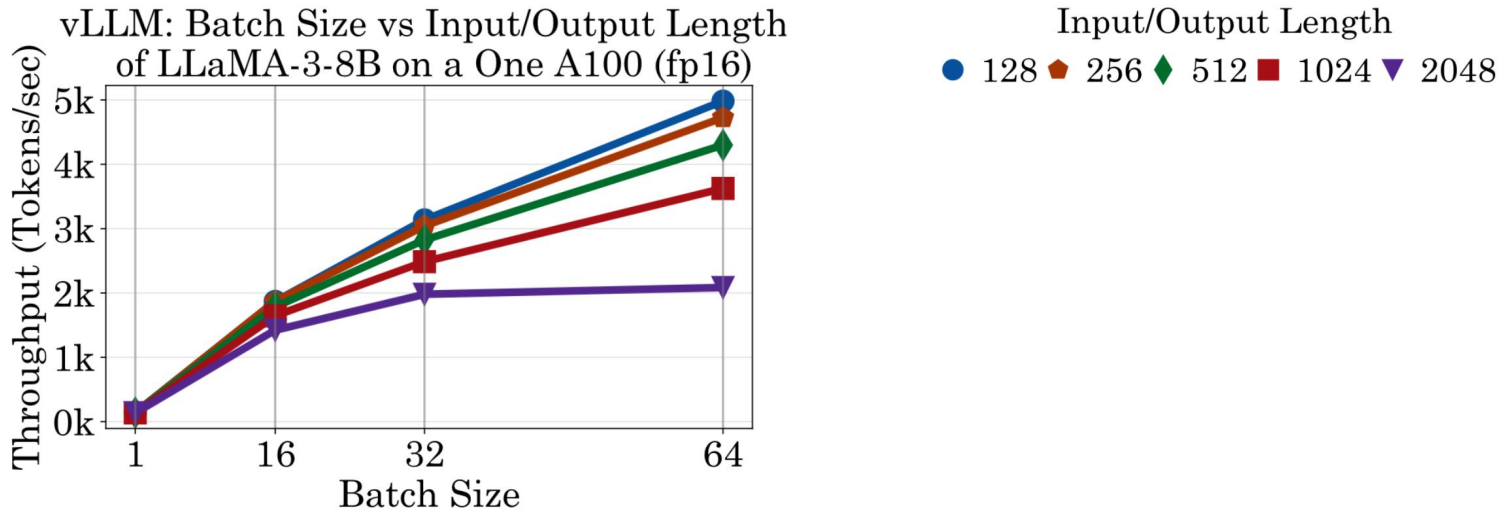


tensor with size $4 \times 3 \times 7$

(order-3 tensor)

BATCHING

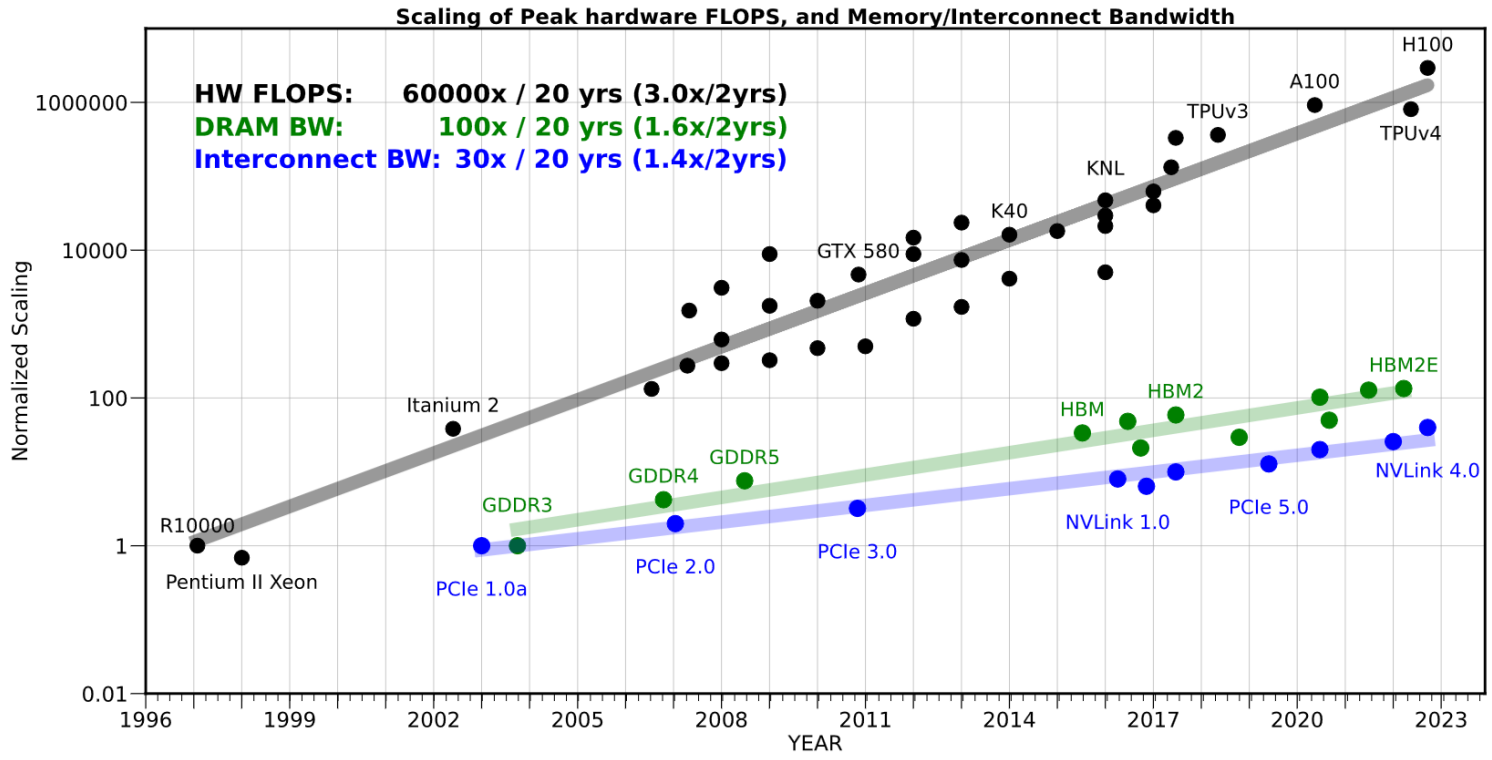
- Once the batch size is sufficiently large, the memory bandwidth of the GPU will become the bottleneck,
- And the overall throughput will stop increasing.



[Chitty-Venkata et al., 2024]

GPU MEMORY IS THE BOTTLENECK

- In GPUs, the processors can process data faster than data can be written to high bandwidth memory (HBM).



[Gholami et al., 2024]

CAN WE MAKE TRANSFORMERS FASTER?

- Since transformers are ubiquitous in NLP today, maybe we can find ways to make transformers faster.
- One way is to optimize the attention mechanism.
- GPU memory is divided into two sections:
  - All GPU cores have shared access to high bandwidth memory (HBM).
  - Each GPU core has exclusive access to static random access memory (SRAM), which is smaller but faster than HBM.
- Example:
  - On an A100, there are 108 cores (i.e., “streaming multiprocessors”).
  - 40 or 80GB of HBM, with a bandwidth of 1.5-2.0 TB/s.
  - 192KB of SRAM per core, with a bandwidth of 19 TB/s.

[Dao et al., 2022]

CAN WE MAKE TRANSFORMERS FASTER?

- Recall that, given matrices K, Q, and V, the output of the attention layer is: (for one head)

$$O = \text{softmax}(QK^T/\sqrt{d})V$$

- Implementing this in PyTorch naively is easy:

```
import math
import torch.nn.functional as F
```

```
attention_scores = query @ key.T
attention_scores = attention_scores / math.sqrt(d_k)
attention_weights = F.softmax(attention_scores, dim=-1)
```

```
context_vector = attention_weights @ value
print(context_vector)
```

CAN WE MAKE TRANSFORMERS FASTER?

- Recall that, given matrices K, Q, and V, the output of the attention layer is: (for one head)

$$O = \text{softmax}(QK^T/\sqrt{d})V$$

- But recall that each of these operations (matrix product, softmax) are executed as GPU kernels.
- GPU kernels are dispatched to GPU cores along with an assigned tile or block of the input.
- The memory needed by that kernel is copied from HBM into the core's SRAM.

CAN WE MAKE TRANSFORMERS FASTER?

- Recall that, given matrices K, Q, and V, the output of the attention layer is: (for one head)

$$O = \texttt{softmax}(QK^\top/\sqrt{d})\,V$$

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

- The naïve implementation involves three kernels:
  - A matrix product, followed by a softmax, followed by matrix product.
  - (there may also be a mask kernel, if using a causal mask)

CAN WE MAKE TRANSFORMERS FASTER?

- Recall that, given matrices K, Q, and V, the output of the attention layer is: (for one head)

$$O = \text{softmax}(QK^\top/\sqrt{d})\,V$$

Algorithm 0 Standard Attention Implementation

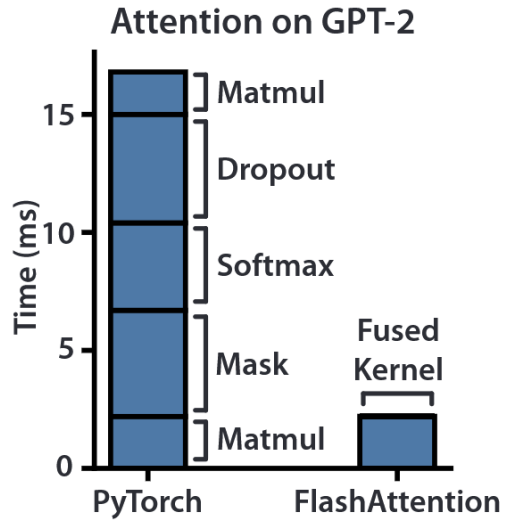
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

- So the naïve implementation requires 3-4 reads from HBM and 3-4 writes to HBM, which is the primary bottleneck.

KERNEL FUSION

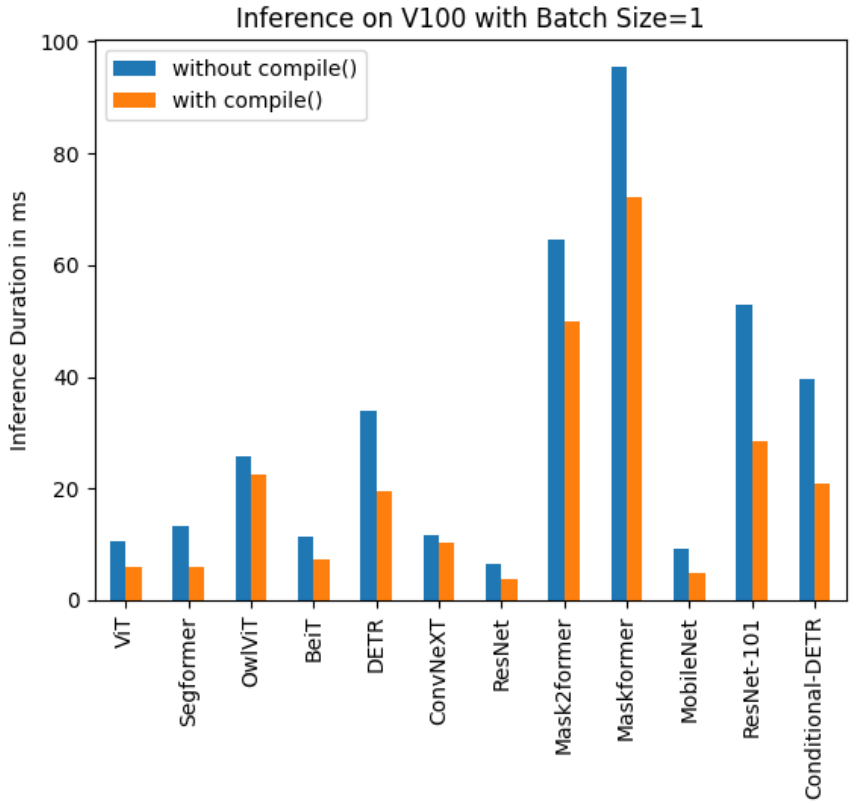
- If we write the attention operation as a single kernel, as opposed to 3-4 separate kernels, we can avoid unnecessary reads/writes from HBM.
  - We can keep the memory in SRAM and the registers of GPU cores for longer.
- The idea of combining multiple distinct kernels into a single kernel is called kernel fusion.



[Dao et al., 2022]

KERNEL FUSION

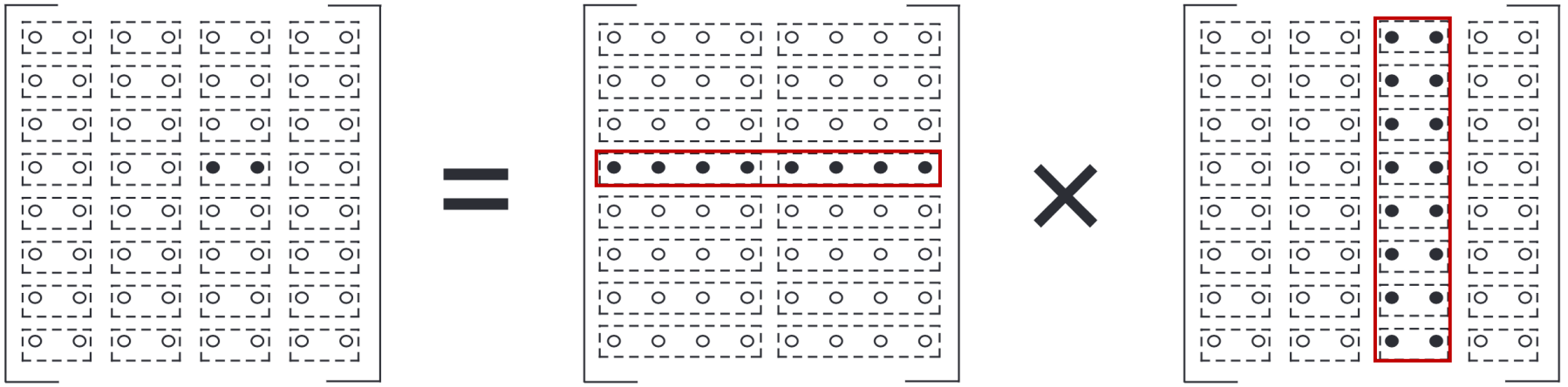
- Kernel fusion can be done automatically.
- PyTorch performs kernel fusion automatically:
  - `torch.compile(...)`
- XLA is a compiler that works with multiple frameworks, including PyTorch and TensorFlow.



[Hugging Face Documentation, Optimize inference using torch.compile()]

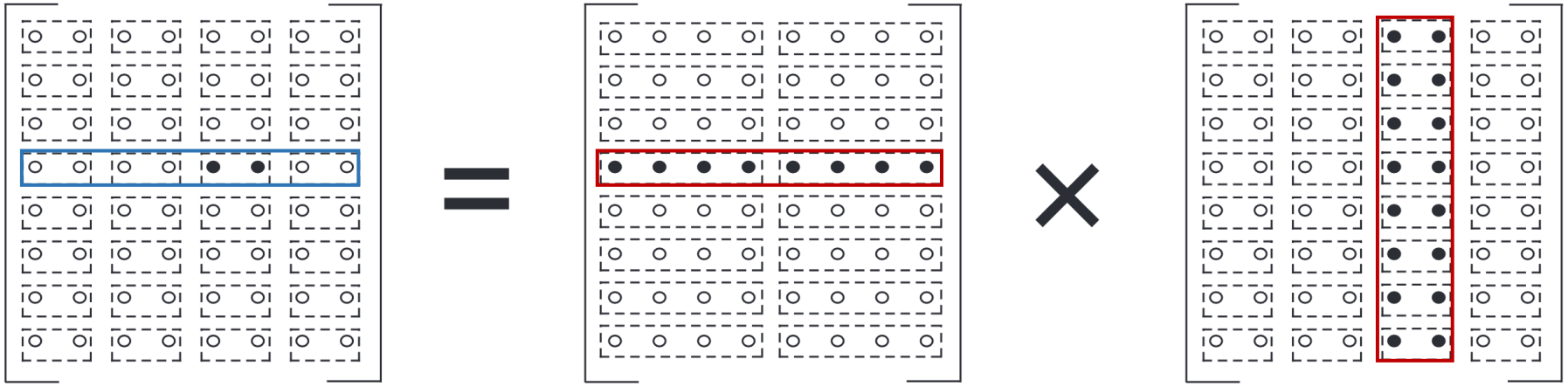
KERNEL FUSION FOR ATTENTION

- But kernel fusion alone doesn't magically get rid of all unnecessary reads/writes to HBM.
- In the first step of the attention computation, we compute the product QK^T.
- We perform this product block-wise, where each GPU core is assigned to a submatrix of QK^T.



KERNEL FUSION FOR ATTENTION

- The next step is to compute the softmax over each row of $QK^T/\sqrt{d}$.
- But notice that the softmax requires access to the full row,
  - Whereas each core only has information about one block/tile.



KERNEL FUSION FOR ATTENTION

- The next step is to compute the softmax over each row of $QK^T/\sqrt{d}$.
- But notice that the softmax requires access to the full row,
  - Whereas each core only has information about one block/tile.
- So how can we divide the softmax operation across GPU cores?

$$\text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

$$m(x) := \max_i \; x_i, \quad f(x) := \begin{bmatrix} e^{x_1 - m(x)} & \dots & e^{x_B - m(x)} \end{bmatrix}, \quad \ell(x) := \sum_i f(x)_i,$$

KERNEL FUSION FOR ATTENTION

- Suppose for simplicity, the row x is divided into two blocks: $x^{(1)}$ and $x^{(2)}$.
- We can compute the numerator f and denominator l for each these blocks.
- After f and l have been computed for both blocks, we can aggregate them to compute the full softmax with the below formulas:

$$\text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

$$m(x) = m(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \begin{bmatrix} e^{m(x^{(1)})-m(x)} f(x^{(1)}) & e^{m(x^{(2)})-m(x)} f(x^{(2)}) \end{bmatrix},$$

$$\ell(x) = \ell(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = e^{m(x^{(1)})-m(x)} \ell(x^{(1)}) + e^{m(x^{(2)})-m(x)} \ell(x^{(2)}),$$

[Dao et al., 2022]

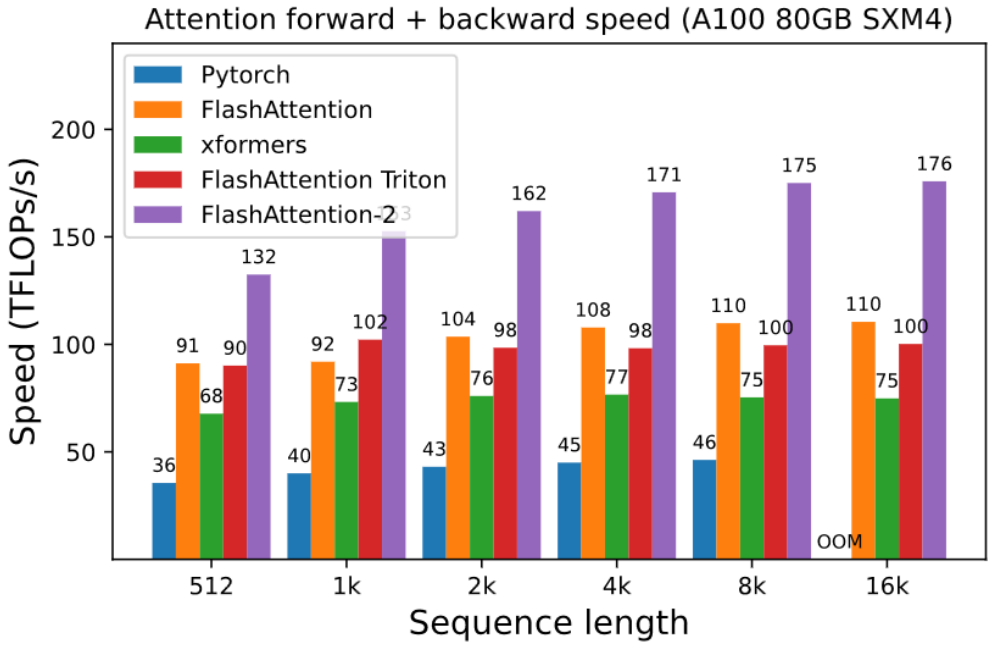
FLASHATTENTION

- After we have computed a block of $\texttt{softmax}(QK^\top/\sqrt{d})$,
  - The same GPU core can perform block matrix multiplication (with the corresponding rows of V) to compute the block of the final output: $\texttt{softmax}(QK^\top/\sqrt{d})V$.
- This method is called FlashAttention (Dao et al., 2022).

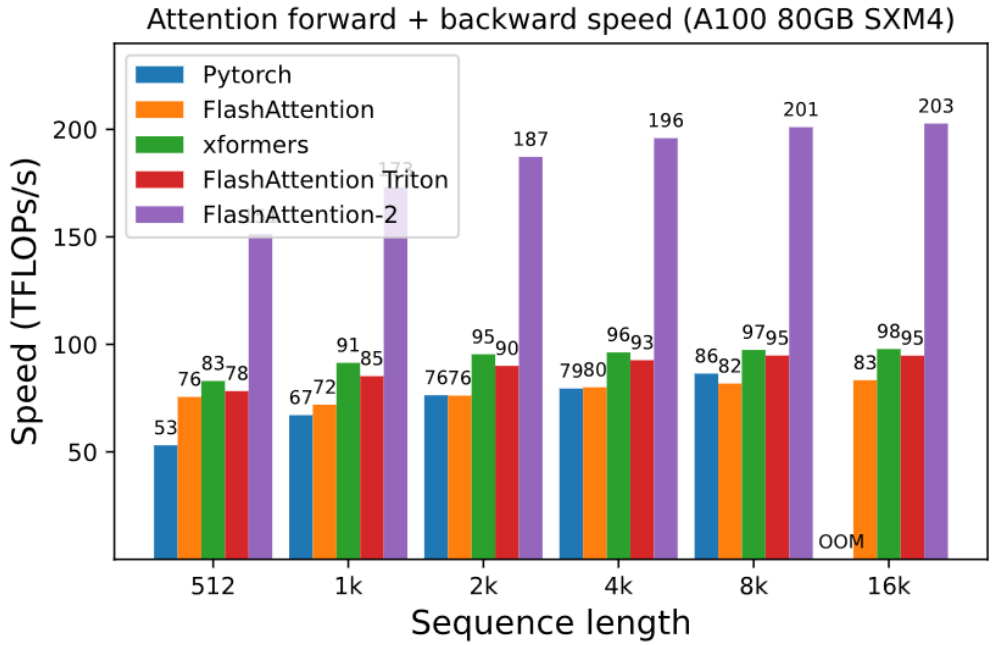
Model implementations	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Huggingface [87]	18.2	9.5 days (1.0×)
GPT-2 small - Megatron-LM [77]	18.2	4.7 days (2.0×)
GPT-2 small - FlashAttention	18.2	**2.7 days (3.5×)**
GPT-2 medium - Huggingface [87]	14.2	21.0 days (1.0×)
GPT-2 medium - Megatron-LM [77]	14.3	11.5 days (1.8×)
GPT-2 medium - FlashAttention	14.3	**6.9 days (3.0×)**

FLASHATTENTION

- Further optimizations have further improved the performance of the kernel.



(a) Without causal mask, head dimension 64

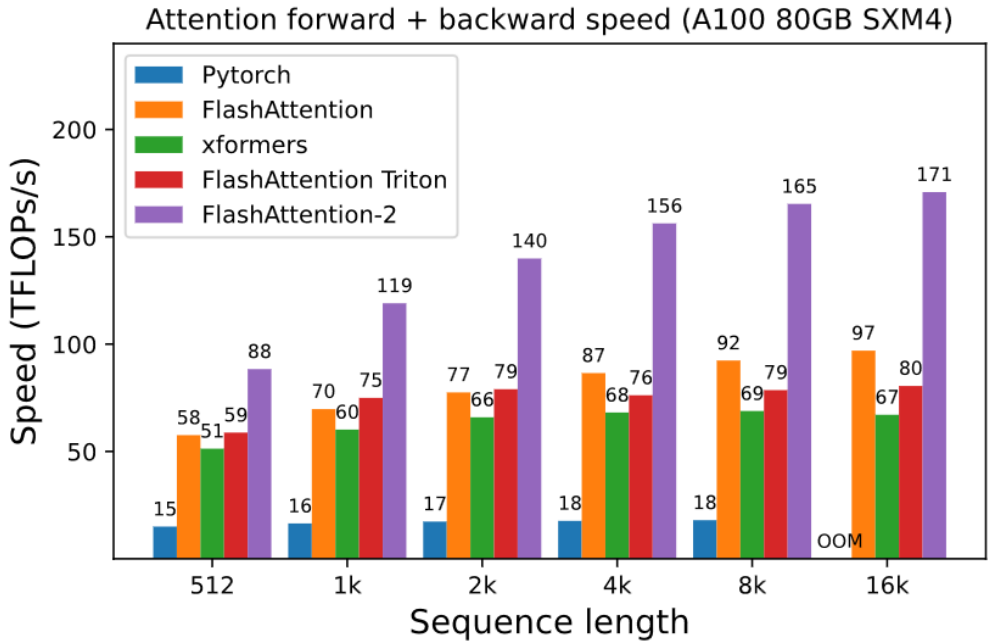


(b) Without causal mask, head dimension 128

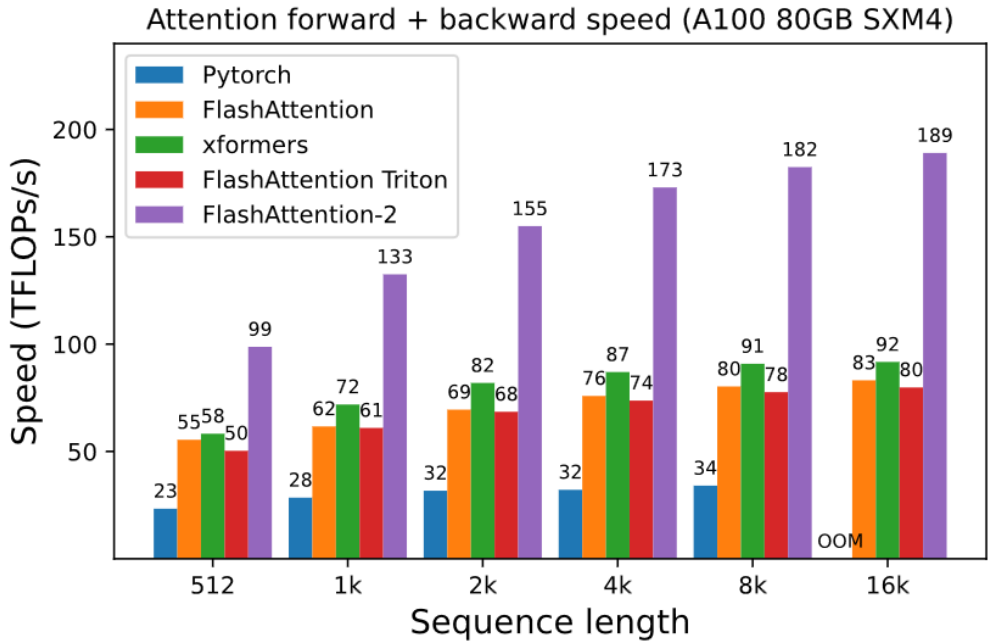
[Dao et al., 2023]

FLASHATTENTION 2

- Further optimizations have further improved the performance of the kernel.



(c) With causal mask, head dimension 64

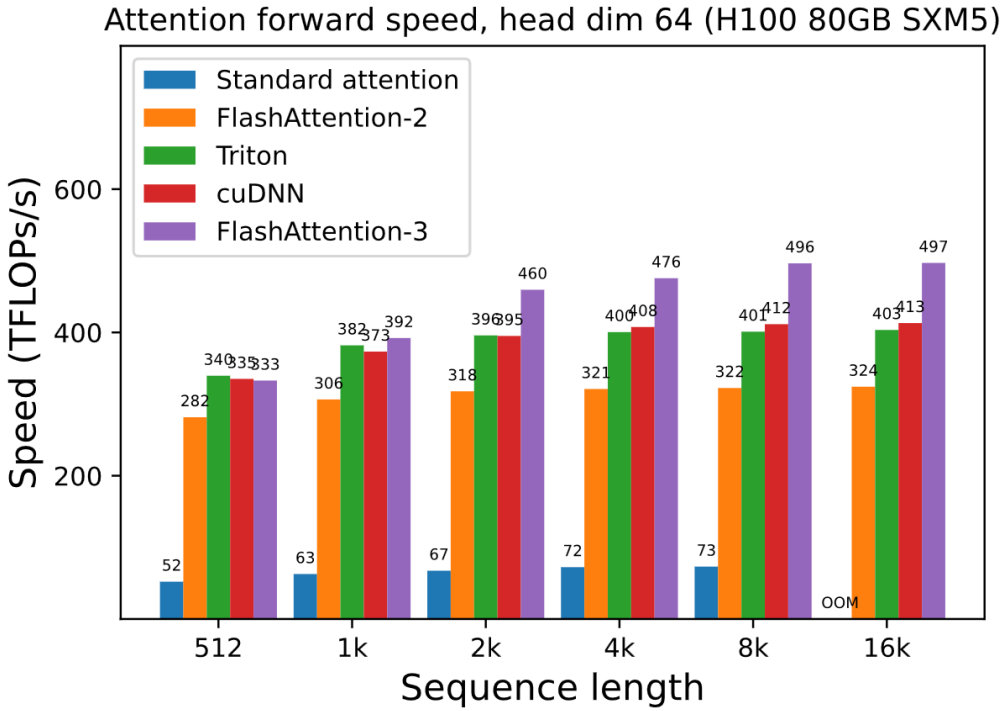


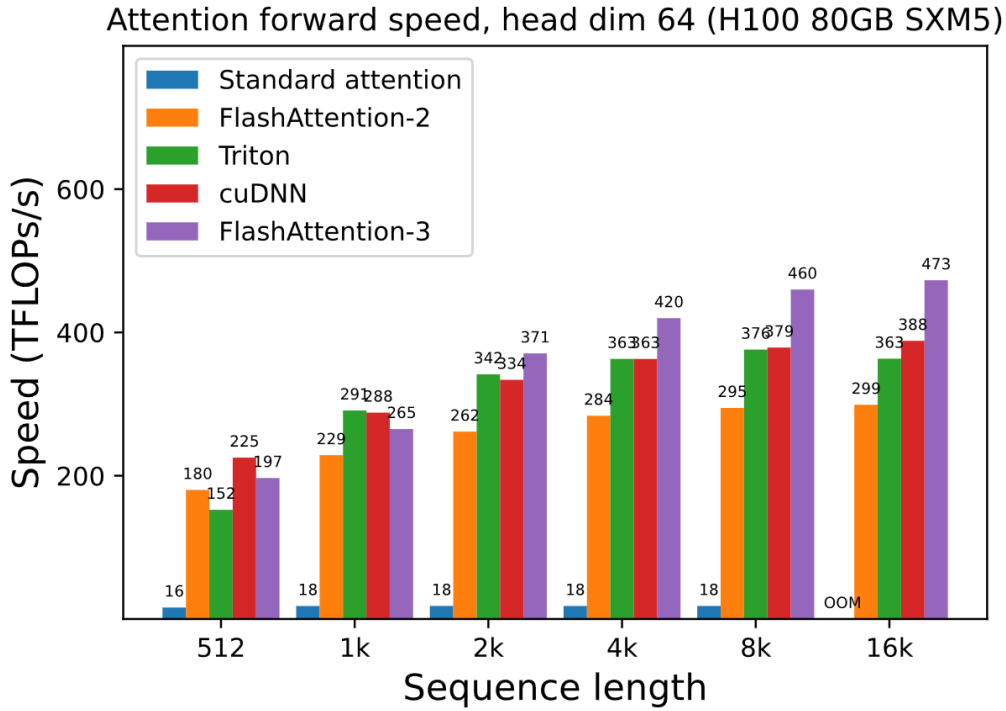
(d) With causal mask, head dimension 128

[Dao et al., 2023]

FLASHATTENTION 3

- Further optimizations have further improved the performance of the kernel.





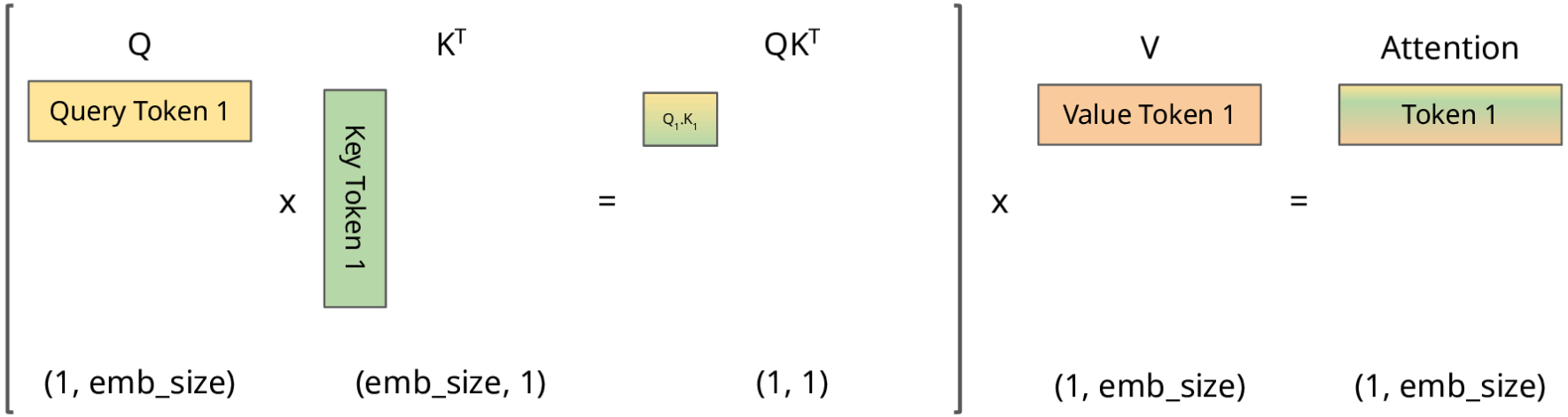
[Shah and Bikshandi et al., 2024]

CAN WE MAKE TRANSFORMERS FASTER?

- We have discussed kernel-level and hardware-aware optimizations that help to speed up transformers.
  - Block matrix multiplication, FlashAttention.
- Are there higher-level optimizations that we can do?
- Let’s focus on decoder-only transformer models.
- Recall that in decoder-only models, a causal mask prevents each token from attending to any token that comes after it.
- During inference, we generate tokens one after the other.
- Is there any redundant computation that we can remove?

CAN WE MAKE TRANSFORMERS FASTER?

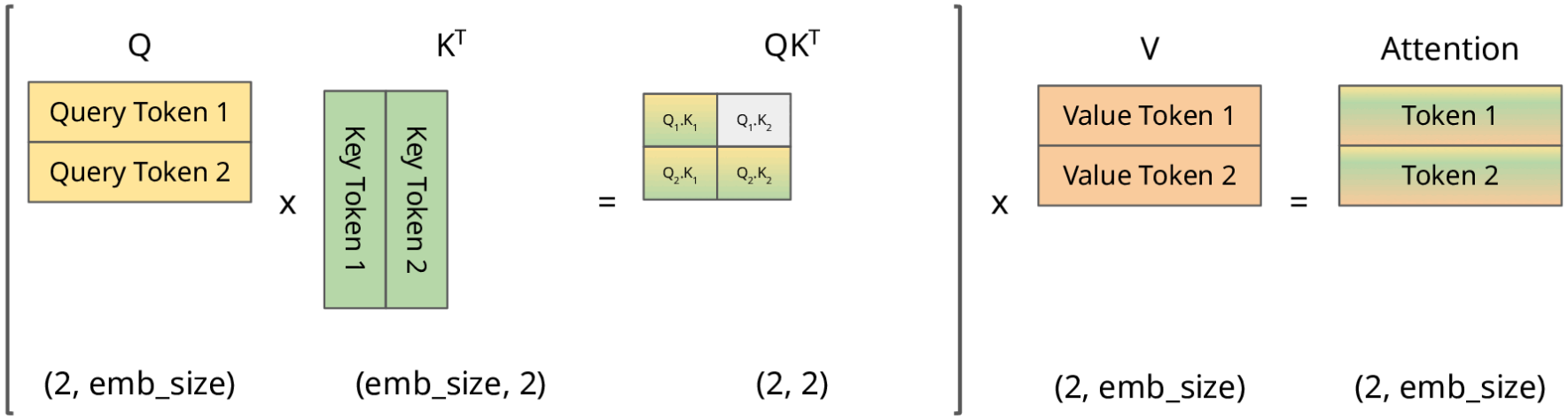
- Consider the attention computation while generating a sequence of 4 tokens:



[João Lages, Transformers KV Caching Explained, 2023]

CAN WE MAKE TRANSFORMERS FASTER?

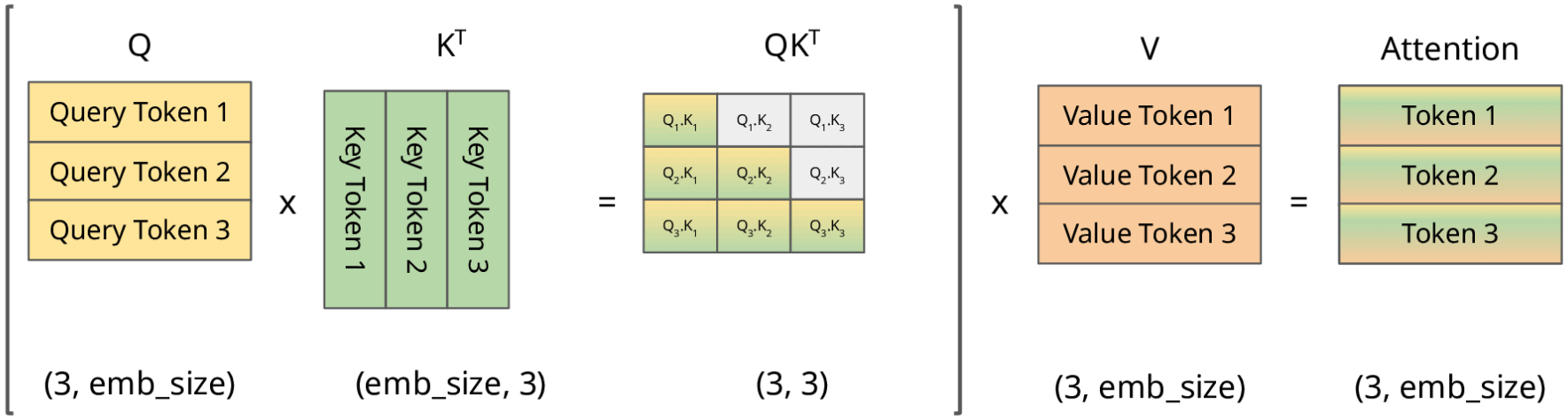
- Consider the attention computation while generating a sequence of 4 tokens:



[João Lages, Transformers KV Caching Explained, 2023]

CAN WE MAKE TRANSFORMERS FASTER?

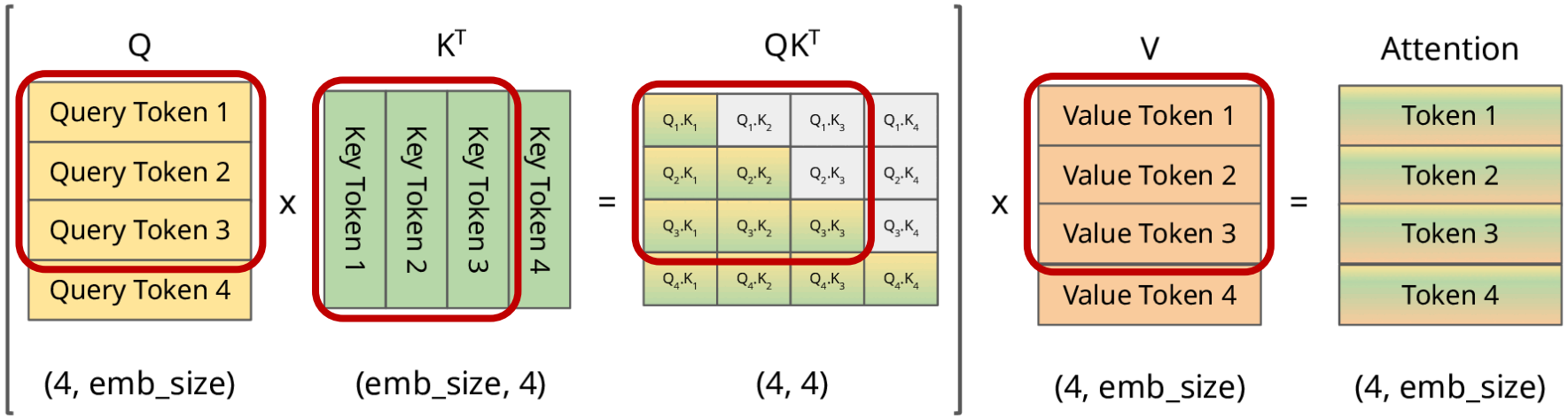
- Consider the attention computation while generating a sequence of 4 tokens:



[João Lages, Transformers KV Caching Explained, 2023]

CAN WE MAKE TRANSFORMERS FASTER?

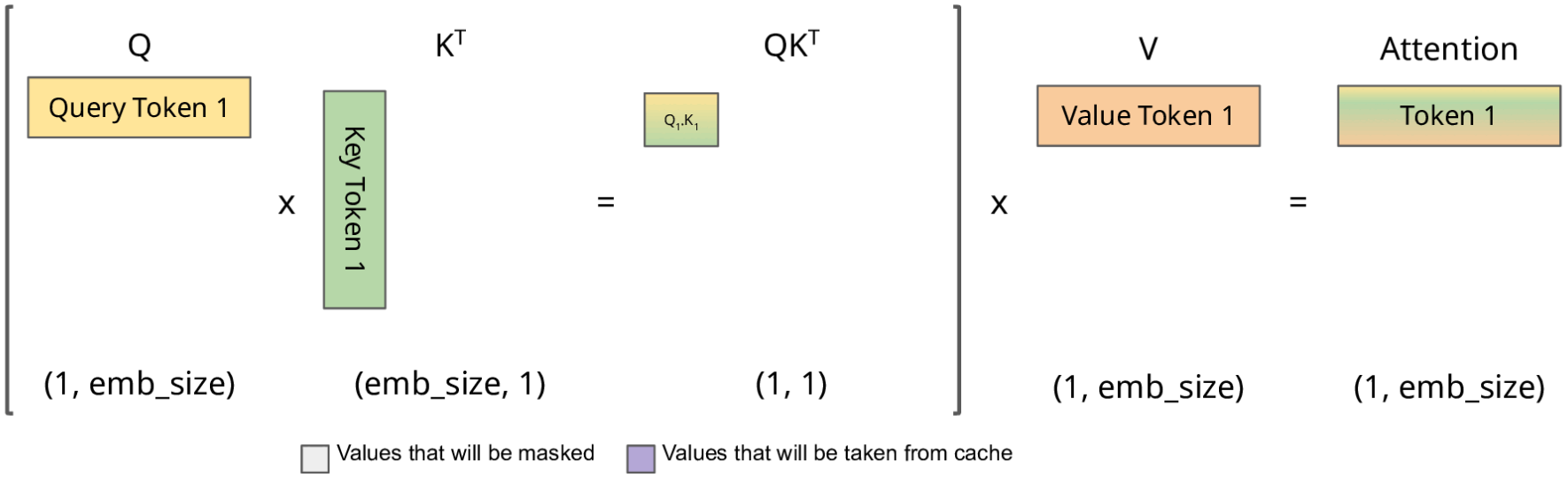
- Consider the attention computation while generating a sequence of 4 tokens:
- Notice that with each forward pass, we are redundantly computing the key, query, and value vectors for all tokens except the last one.



[João Lages, Transformers KV Caching Explained, 2023]

CAN WE MAKE TRANSFORMERS FASTER?

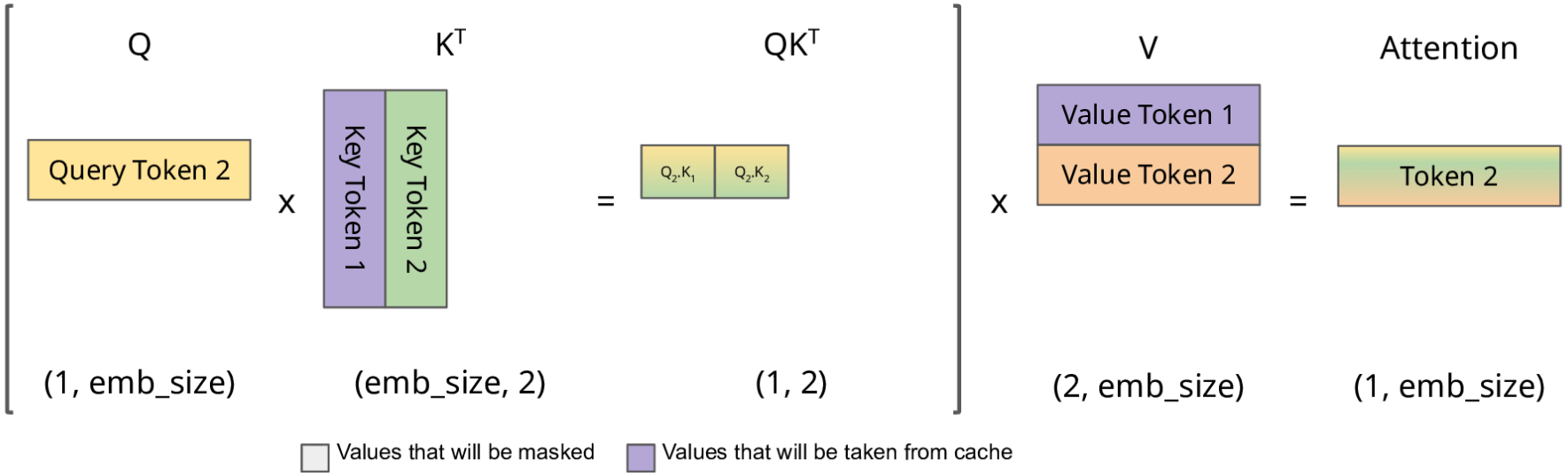
- We can instead store these old key and value vectors in a cache, and instead compute the query vector for only the latest token.



[João Lages, Transformers KV Caching Explained, 2023]

KEY-VALUE CACHING

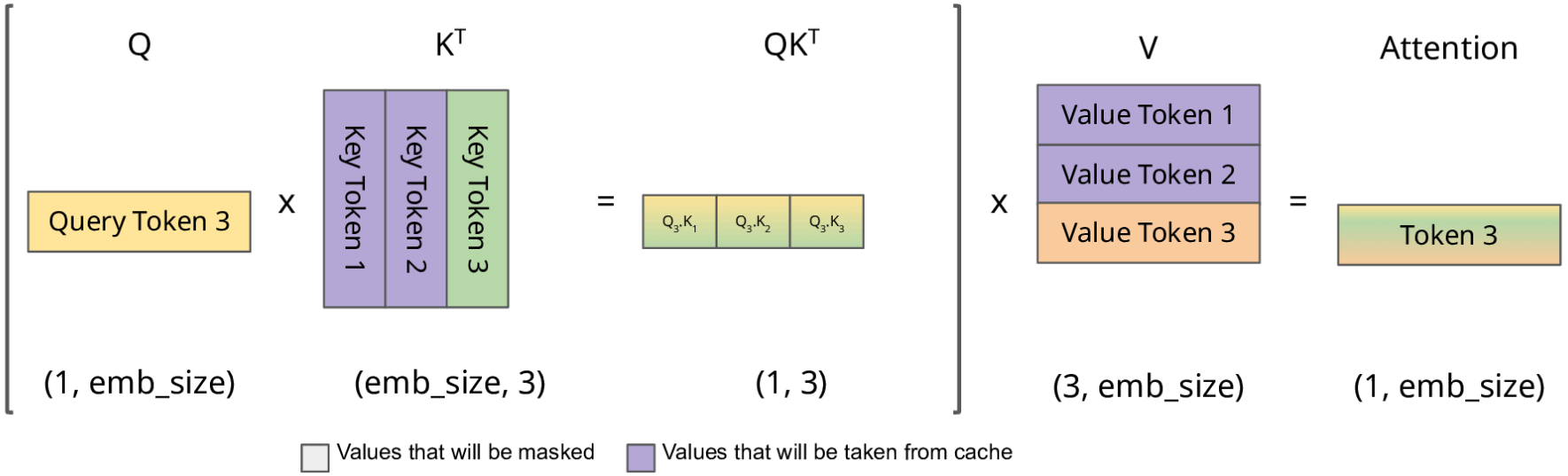
- We can instead store these old key and value vectors in a cache, and instead compute the query vector for only the latest token.



[João Lages, Transformers KV Caching Explained, 2023]

KEY-VALUE CACHING

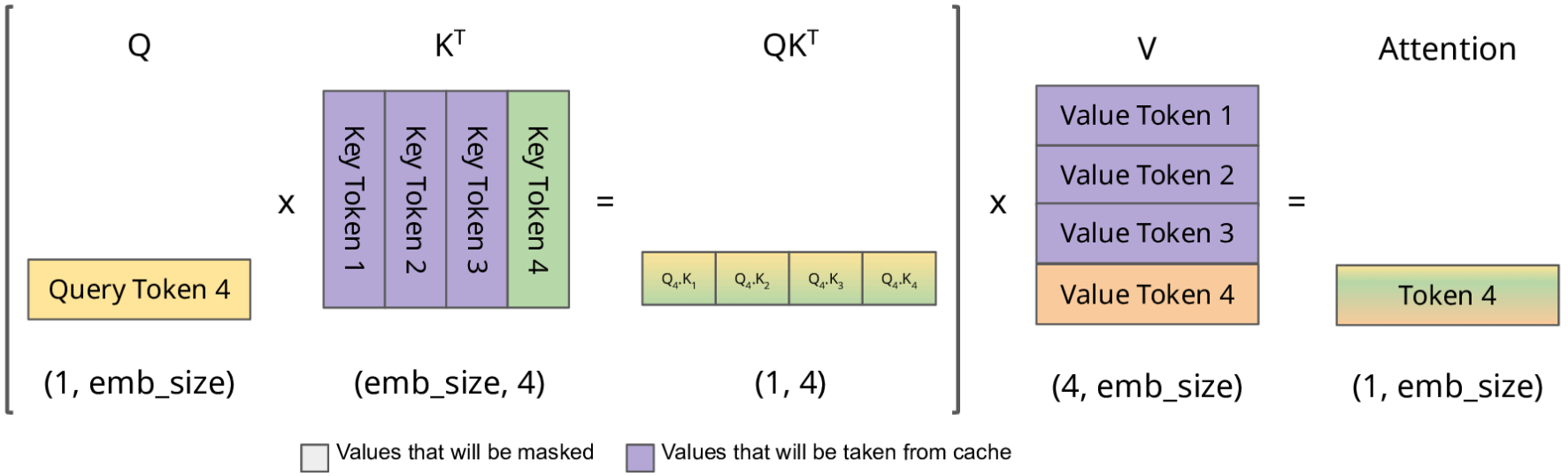
- We can instead store these old key and value vectors in a cache, and instead compute the query vector for only the latest token.



[João Lages, Transformers KV Caching Explained, 2023]

KEY-VALUE CACHING

- We can instead store these old key and value vectors in a cache, and instead compute the query vector for only the latest token.



[João Lages, Transformers KV Caching Explained, 2023]

KEY-VALUE CACHING

- This approach is called key-value caching (KV caching).
- Since this method relies on tokens in a sequence being generated one after the other, the KV cache is only used during inference.
- KV caching only works for decoder-only models (i.e., with a causal mask).
- KV caching reduces the cost of the forward pass from $O(n^2d^2)$ to $O(nd^2)$ where n is the sequence length and d is the model dimension.
- Some benchmarks: Perform 10 generation steps with GPT-2 on Tesla T4 GPU.
  - Without KV caching: 56.197 ± 1.855 seconds
  - With KV caching: 11.885 ± 0.272 seconds

[João Lages, Transformers KV Caching Explained, 2023]

KEY-VALUE CACHING

- But the KV cache requires additional memory to store the cache.
  - For each attention layer, and for each attention head, we must store n key and value vectors, each with dimension d.
  - So the total memory requirement is: $2BndH$.
    - Where B is batch size and H is number of attention heads.

REDUCING MEMORY FOOTPRINT OF KV CACHE

- Can we reduce the memory requirements of KV caching?
- In standard multi-head attention, for each attention layer,
  - We compute H query vectors, H key vectors, and H value vectors.
- What if each attention head did not have its own key and value vectors?
  - What if we only compute a single key vector and a single value vector to use across all attention heads?
  - So the only difference between attention heads is the query vector.
- This approach is called multi-query attention (MQA; Shazeer, 2019).
- Note that this is an architectural modification of the transformer.
- So the model needs to be trained using MQA.

MULTI-QUERY ATTENTION

- MQA reduces the expressivity of the standard (MHA) transformer.
- There is a cost in how well the model can fit to data (i.e., accuracy).

Table 3: Billion-Word LM Benchmark Results.

Attention	h	d_k, d_v	d_{ff}	dev-PPL
multi-head	8	128	8192	**29.9**
multi-query	8	128	9088	30.2
multi-head	1	128	9984	31.2
multi-head	2	64	9984	31.1
multi-head	4	32	9984	31.0
multi-head	8	16	9984	30.9

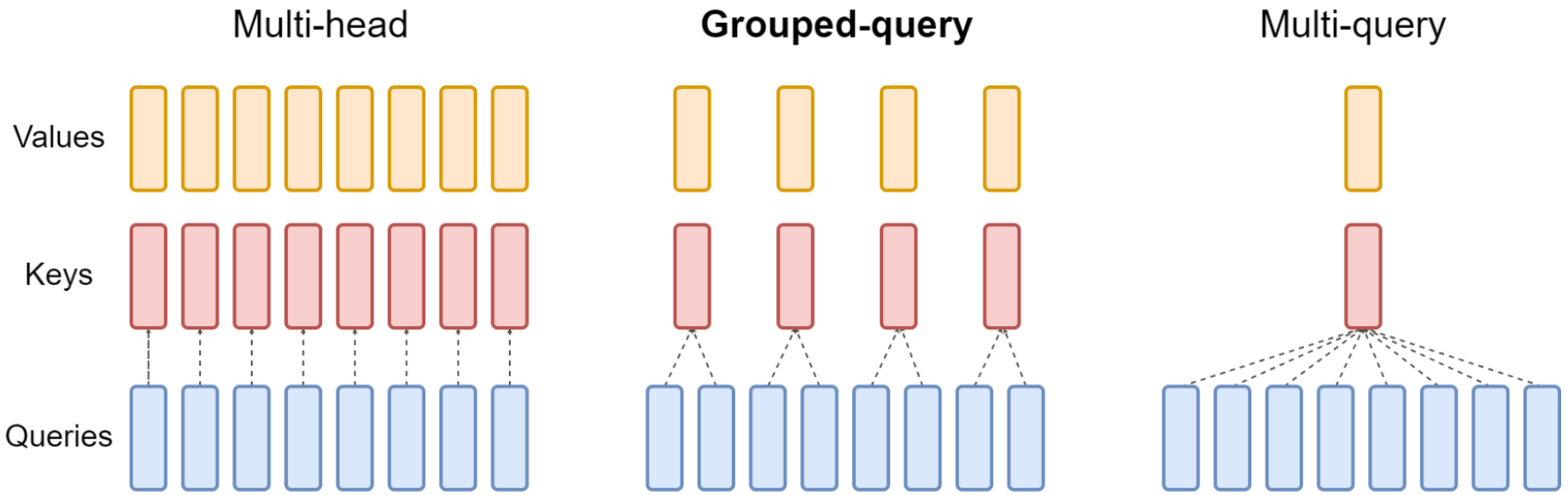
[Shazeer, 2019]

MULTI-QUERY ATTENTION

- Is there some middle-ground between MQA and MHA that is not as memory-intensive as MHA but is more accurate than MQA?
- In MHA, we have a $1:1$ correspondence between key/value vectors and attention heads.
- In MQA, we have a $1:H$ correspondence between key/value vectors and attention heads.
- What if we instead have a $1:r$ correspondence where r is between 1 and H?
  - This is analogous to grouping the H attention heads into H/r groups.
  - Each group of attention heads computes a single key and value vector.

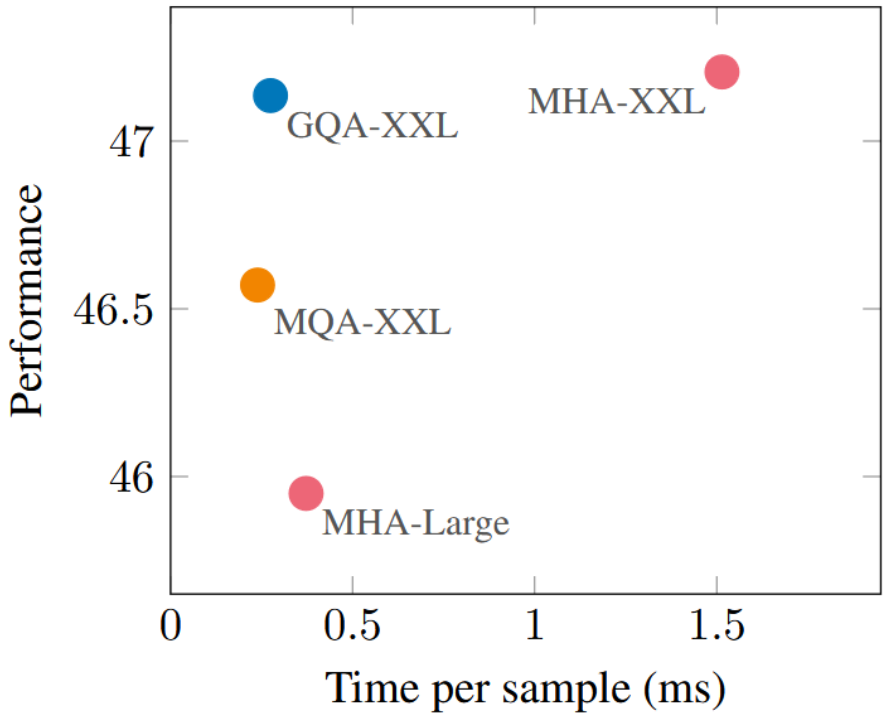
GROUPED QUERY ATTENTION

- This approach is called grouped query attention (GQA; Ainslie, Lee-Thorp, de Jong et al., 2023).



GROUPED QUERY ATTENTION

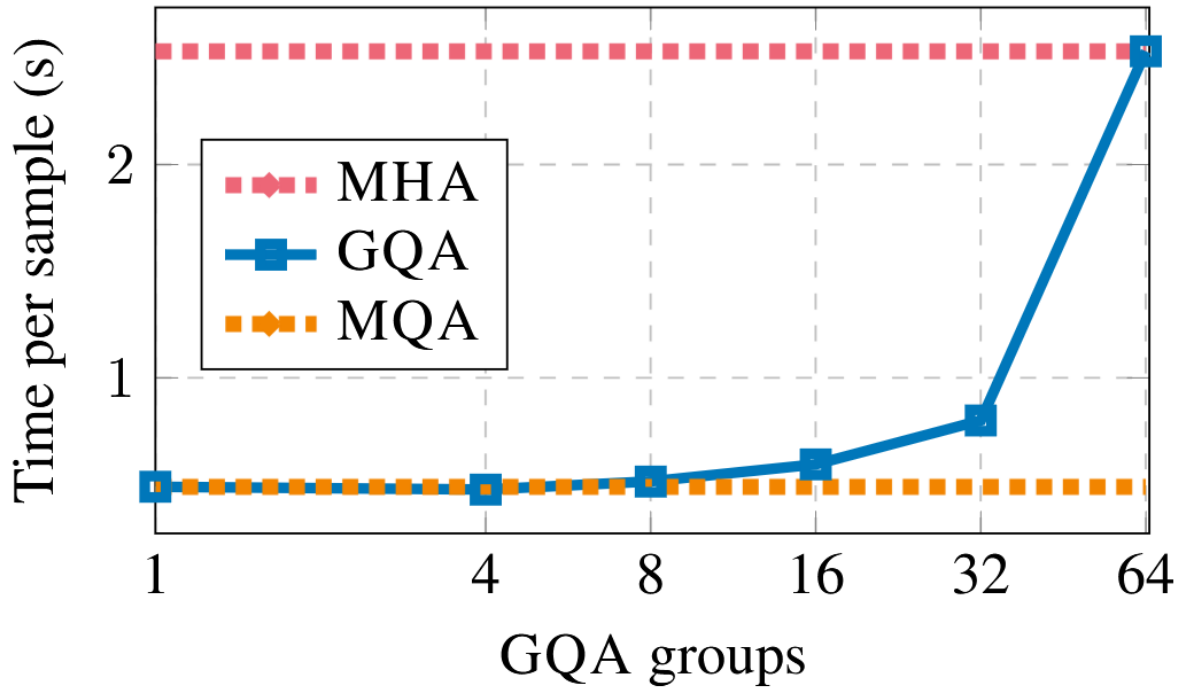
- GQA is similar to MQA in terms of inference speed, but more similar to MHA in terms of accuracy.



[Ainslie, Lee-Thorp, de Jong et al., 2023]

GROUPED QUERY ATTENTION

- GQA is similar to MQA in terms of inference speed, but more similar to MHA in terms of accuracy.



[Ainslie, Lee-Thorp, de Jong et al., 2023]

NEXT: WORKING WITH LARGE MODELS

- We discussed how to make transformers faster,
  - For both training and inference.
- But memory is often a limiting factor.
  - What do we do if a model is larger than the available GPU memory?
  - What if the batch during training won't fit in memory?
  - Different ways to parallelize training and inference of large models.
- Can we trade model size for accuracy?
  - Model compression:
    - Parameter efficient fine-tuning (PEFT) such as LoRA
    - Quantization
    - Distillation

QUESTIONS?