CS 577: NATURAL LANGUAGE PROCESSING

Abulhair Saparov

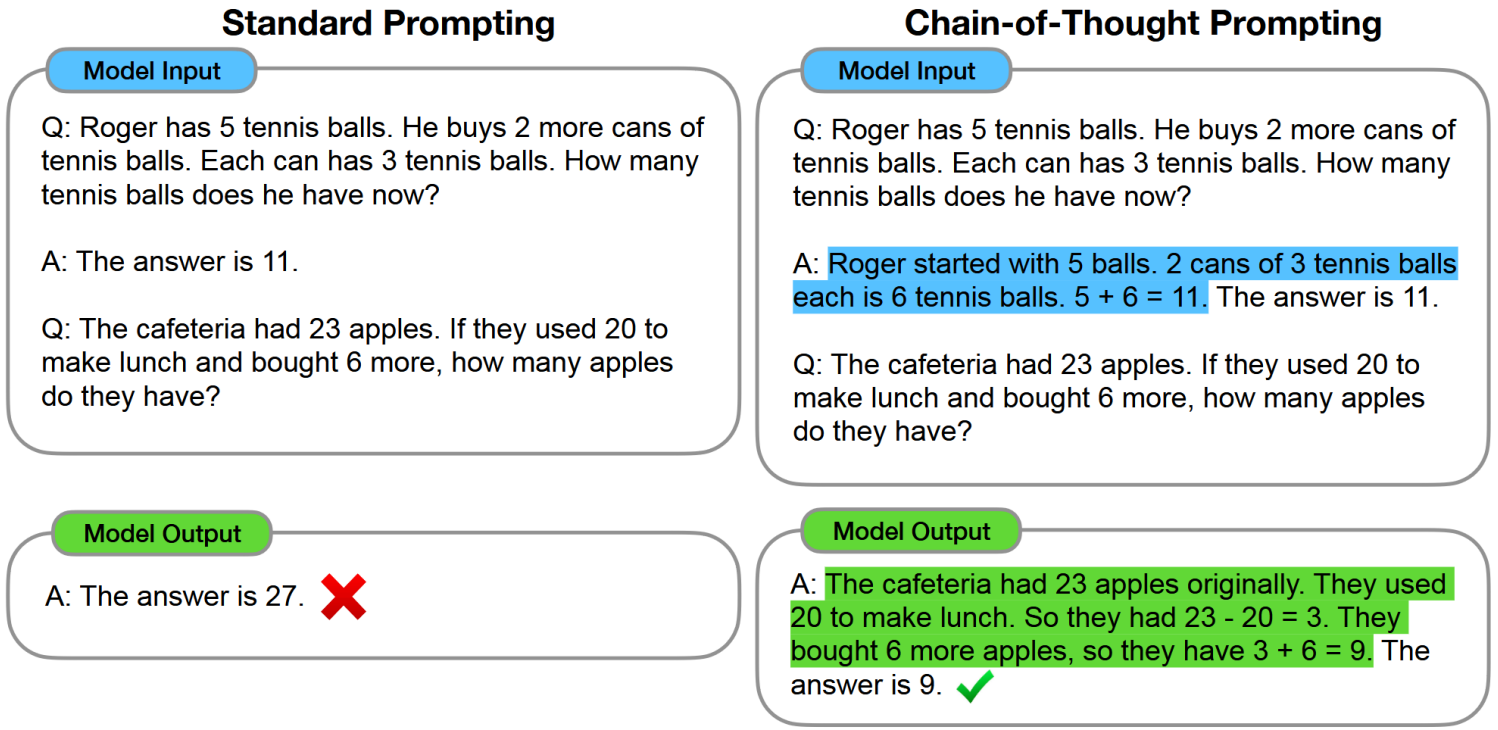
Lecture 10: Efficiency

PREVIOUSLY: PROMPTING

- Last lecture, we discussed how model performance is sensitive to the prompt.
- Perturbations such as spacing, newlines, paraphrasing, etc. can cause significant differences in model accuracy/performance.
- Few-shot prompting and in-context learning can help improve task accuracy.
  - The model is still sensitive to things like:
  - Number of few-shot examples
  - The order of the few-shot examples
  - The diversity/coverage of the few-shot examples

CHAIN-OF-THOUGHT PROMPTING

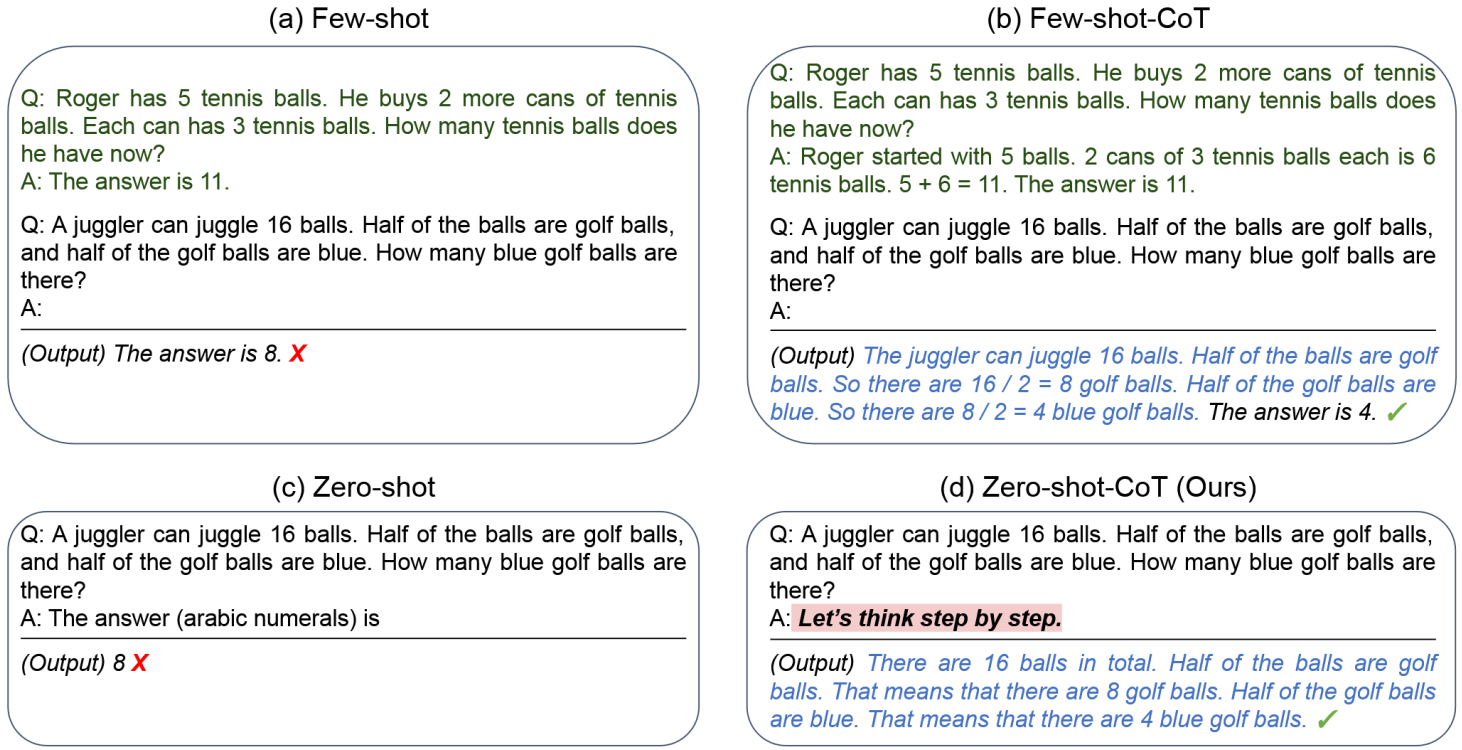
- We can design the prompt to induce the model to output its step-by-step reasoning.



[Wei et al., 2022]

CHAIN-OF-THOUGHT PROMPTING

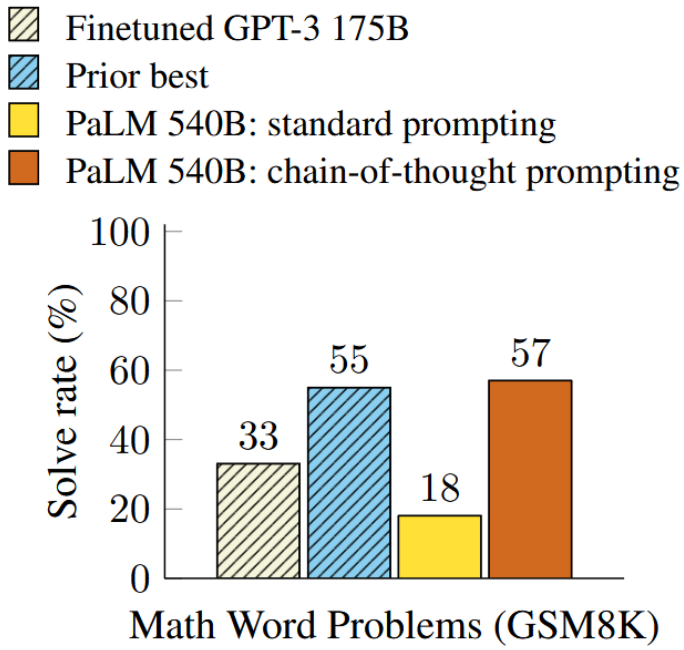
- This is called chain-of-thought (CoT) prompting.
- CoT prompting can be done zero-shot:



[Kojima et al., 2022]

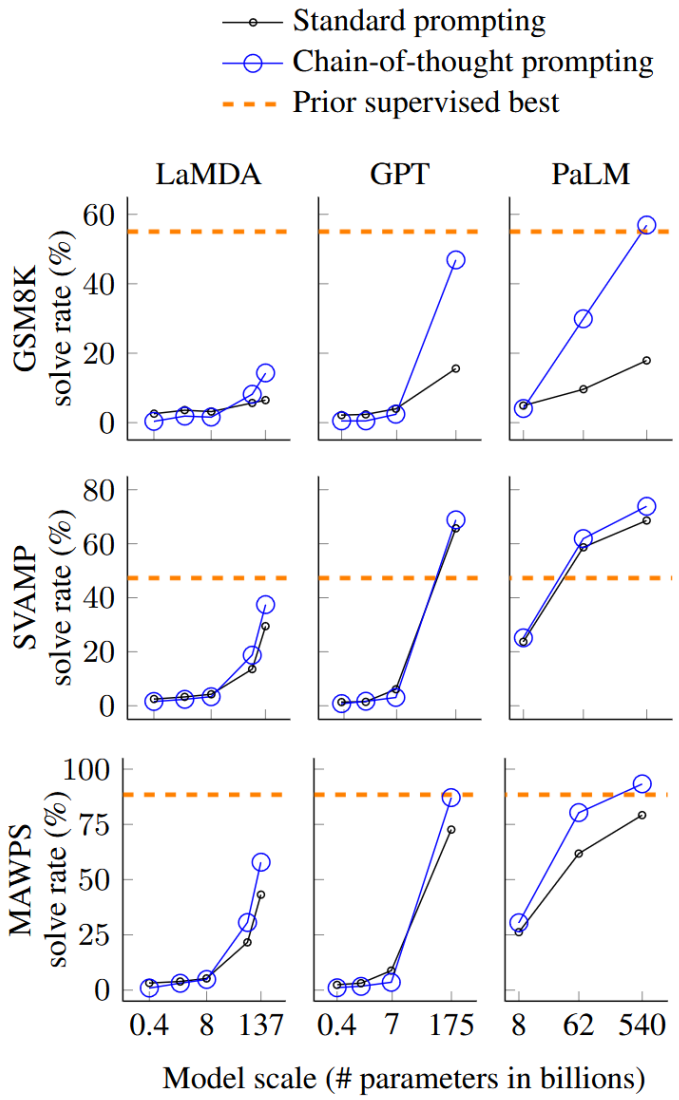
CHAIN-OF-THOUGHT PROMPTING

- CoT prompting can significantly improve performance on reasoning tasks.



[Wei et al., 2022]

CHAIN-OF-THOUGHT PROMPTING

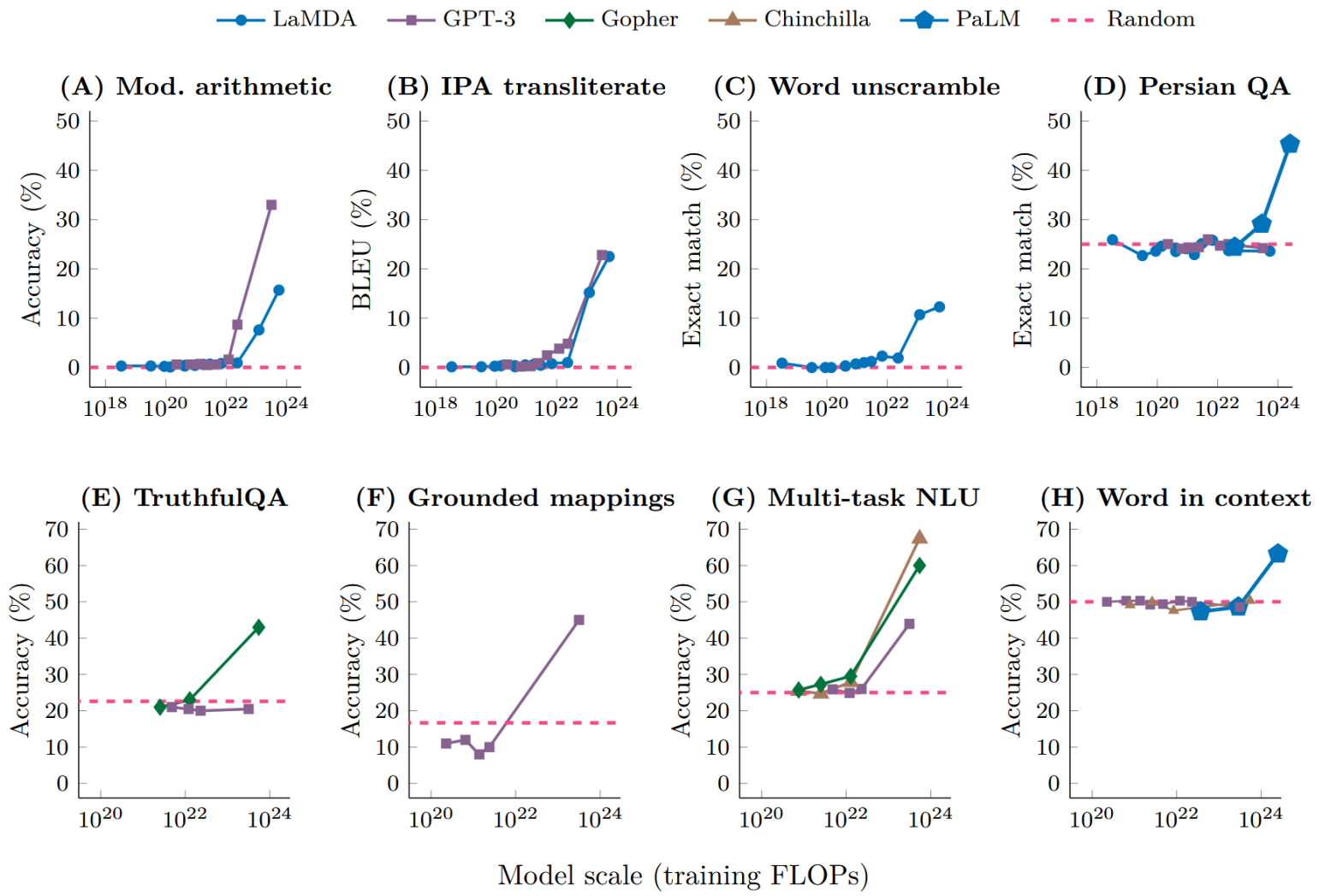


[Wei et al., 2022]

EMERGENT ABILITIES

- Only larger models seem to benefit from few-shot prompting (are capable of in-context learning) or benefit from chain-of-thought prompting.
- Is this a consequence of scaling?
- If so, shouldn't scaling laws predict this ability?
- Few-shot prompting, ICL, and CoT seem to "emerge" suddenly in sufficiently large models.
- Cross-entropy still seems to be decreasing following a power law.
  - There is no sudden/unexpected "drop" in the loss function.

EMERGENT ABILITIES

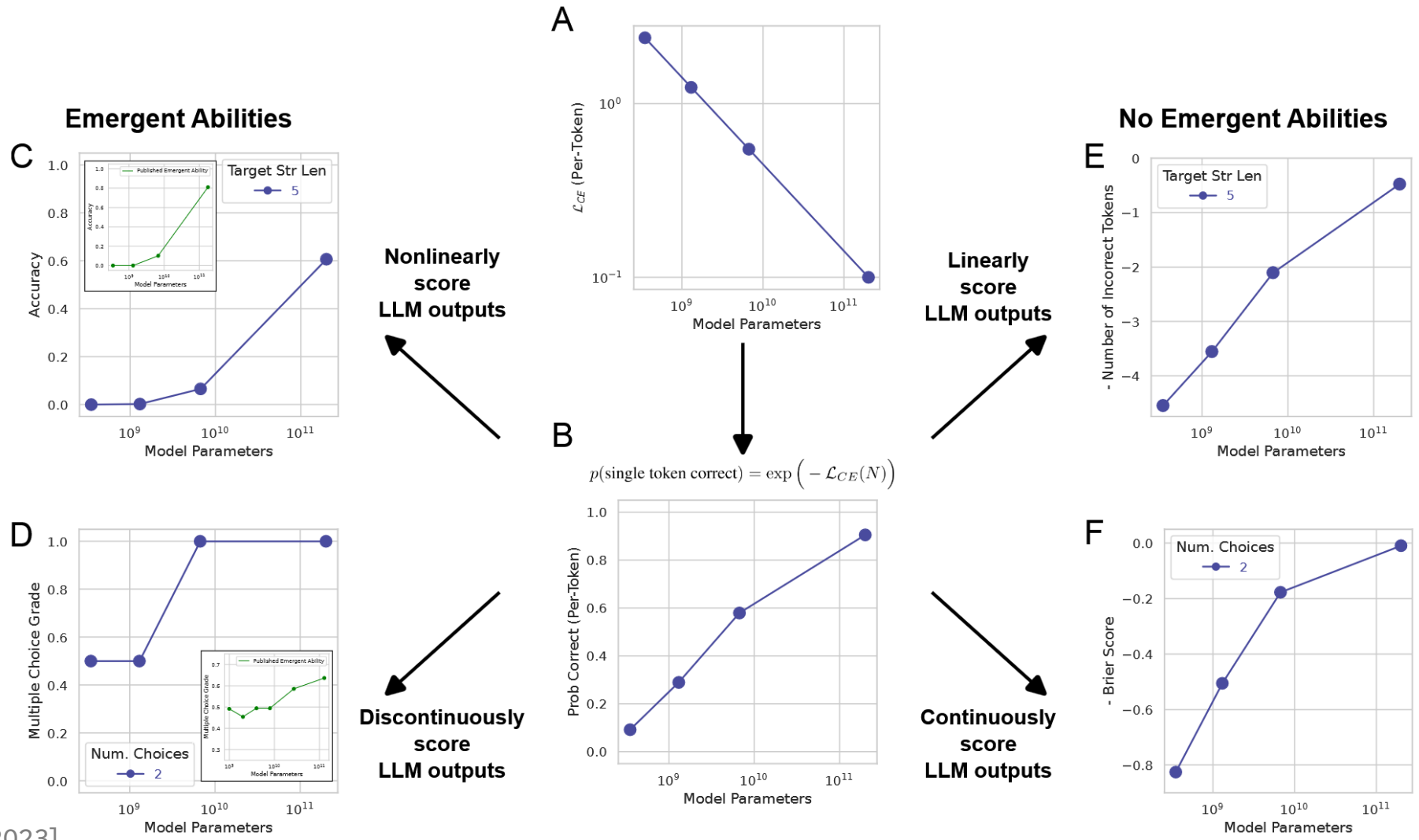


[Wei et al., 2023]

EMERGENT ABILITIES?

- Schaeffer et al. (2023) showed that if you change the performance metric, the “sudden” increase in performance becomes much smoother/linear.

EMERGENT ABILITIES?



[Schaeffer et al., 2023]

EMERGENT ABILITIES?

- Schaeffer et al. (2023) showed that if you change the performance metric, the "sudden" increase in performance becomes much smoother/linear.
- "Emergent abilities" become predictable when using a different metric.
- For example, consider a task that requires performing 10 reasoning steps.
- Suppose the model's probability of correctly performing 1 step increases linearly with scale.
- Then the probability that the model will perform all 10 steps correctly will increase exponentially.
- So overall accuracy will seem sudden/emergent, even if per-step accuracy improves predictably.

EFFICIENCY

COST OF TRAINING AND INFERENCE

- How expensive is it to train and use ML models?
- Consider the forward pass of a model (e.g., transformer, RNN, etc).
- The forward pass consists of multiple operations of different types:
  - Matrix multiplication
  - Softmax
  - Vector addition
  - Vector-scalar operations
  - Activation functions
  - etc...

COST OF VECTOR OPERATIONS

- Some operations, such as vector operations, are relatively cheap.
- Adding two vectors of dimension d takes d floating-point operations.
- Multiplying a d-dimensional vector with a scalar (or dividing) takes d FLOPs.
- What about memory?
  - If computing $f(x) = x + b$, where b is a learnable vector parameter,
  - There is a total of d parameters we need to store in memory.

MATRIX MULTIPLICATION

- Matrix multiplication is significantly more expensive.
- Consider multiplying two matrices $A \in \mathbb{R}^{n \times k}$ and $B \in \mathbb{R}^{k \times m}$.

$$[AB]_{ij} = \sum_{u=1}^{k} A_{iu} B_{uj}.$$

- We must compute each element in the destination matrix.
  - The destination matrix AB has dimension $n \times m$.
  - Each element is computed using a dot product with length k.
  - The dot product requires k multiplications, followed by k additions.
- Thus the total cost (in terms of time) is: $2 \cdot n \cdot k \cdot m$.
- For memory: Suppose we have $f(X) = AX$ where A is learnable.
  - We need to store $n \cdot k$ parameters in memory.

SOFTMAX

- Transformers perform a softmax operation in every attention layer.
  - Each layer has H softmax operations, where H is the number of attention heads.

$$f(\alpha)_k = \frac{\exp(\alpha_k)}{\sum_{j=1}^{K} \exp(\alpha_j)}.$$

- Suppose α is has dimension d.
- Softmax requires computing d exponential operations,
  - Followed by a sum (d operations),
  - Followed by d division operations.
- Softmax requires no additional memory.

SOFTMAX

- Transformers perform a softmax operation in every attention layer.
  - Each layer has H softmax operations, where H is the number of attention heads.

$$f(\alpha)_k = \frac{\exp(\alpha_k)}{\sum_{j=1}^{K} \exp(\alpha_j)}.$$

- The exponential function is expensive to compute.
- The exponential function is transcendental,
  - Meaning we can not write it down as a finite sequence of addition, subtraction, multiplication, or division operations.
  - It is approximated by computing terms in an infinite series.

ACTIVATION FUNCTIONS

- The cost of an activation function depends on the choice of function.
- Some activation functions require the computation of transcendental functions,
  - E.g., sigmoid, tanh.
  - $GELU(x) = \Phi(x) \cdot x$ *where* Φ *is the CDF for the normal distribution.*
  - $Swish(\mathrm{x}) = \sigma(kx) \cdot x$ where k is a learnable parameter.
- Others are much cheaper:
  - $ReLU(\mathrm{x}) = \mathbb{1}\{x > 0\} \cdot x$
- These functions cost d operations,
- But each operation may be more expensive due to transcendental functions.
- Trivia: GPT-2 used GELU.

ACTIVATION FUNCTIONS

- The cost of an activation function depends on the choice of function.
- Others contain matrix multiplications:
  - $GLU(\mathrm{x}) = \sigma(W_1 x + b_1) \odot (W_2 x + b_2)$ where W_1, W_2, b_1, b_2 are learnable.
  - $SwiGLU(\mathrm{x}) = Swish(W_1 x + b_1) \odot (W_2 x + b_2)$
- Due to the additional matrix products, these functions cost $4d^2$.
- Trivia: Llama-2, Llama-3, Deepseek, OLMO, PaLM use SwiGLU.
- For memory, the simpler activation functions require no additional parameters.
  - But SwiGLU requires $2(d^2 + d)$ parameters.

COST OF FORWARD PASS

- Given the cost of each component, we can now compute the total cost of the forward pass for a given ML model.
- Let's take the decoder-only transformer for example.

Operation	Parameters	FLOPs per Token
Embed	$(n_{\text{vocab}} + n_{\text{ctx}})d_{\text{model}}$	$4d_{\text{model}}$
Attention: QKV	$n_{\text{layer}}d_{\text{model}}3d_{\text{attn}}$	$2n_{\text{layer}}d_{\text{model}}3d_{\text{attn}}$
Attention: Mask	—	$2n_{\text{layer}}n_{\text{ctx}}d_{\text{attn}}$
Attention: Project	$n_{\text{layer}}d_{\text{attn}}d_{\text{model}}$	$2n_{\text{layer}}d_{\text{attn}}d_{\text{model}}$
Feedforward	$n_{\text{layer}}2d_{\text{model}}d_{\text{ff}}$	$2n_{\text{layer}}2d_{\text{model}}d_{\text{ff}}$
De-embed	—	$2d_{\text{model}}n_{\text{vocab}}$
Total (Non-Embedding)	$N = 2d_{\text{model}}n_{\text{layer}}(2d_{\text{attn}} + d_{\text{ff}})$	$C_{\text{forward}} = 2N + 2n_{\text{layer}}n_{\text{ctx}}d_{\text{attn}}$

Note: These are matrix products

[Adam Casson, Transformer FLOPs, 2023]

COST OF FORWARD PASS

- Total number of non-embedding parameters for decoder-only transformer:

$$N = 2d_{model}n_{layer}(2d_{attn} + d_{ff}).$$

- Total number of FLOPs per token for the forward pass of a decoder-only transformer:

$$C_{forward} = 4d_{model}n_{layer}(2d_{attn} + d_{ff}) + 2n_{layer}n_{ctx}d_{attn},$$
$$= 2N + 2n_{layer}n_{ctx}d_{attn}.$$

- In most modern LLMs, $d_{model} = d_{attn}$, and d_{ff} is set to a multiple of d_{model} ($4d_{model}$ is a common choice).
- Assuming these choices, we can re-write the above:

$$N = 2d_{model}n_{layer}(2d_{model} + 4d_{model}) = 2d_{model}n_{layer}(6d_{model}) = 12d_{model}^2n_{layer},$$
$$C_{forward} = 2N + 2n_{layer}n_{ctx}d_{model}.$$

[Adam Casson, Transformer FLOPs, 2023]

COST OF FORWARD PASS

$$N = 12 d_{model}^2 n_{layer},$$

$$C_{forward} = 2N + 2 n_{layer} n_{ctx} d_{model}.$$

- Note that N *depends on the square of* d_{model}, *whereas the term* $2 n_{layer} n_{ctx} d_{model}$ is linear in d_{model}.
- Thus, if d_{model} is large, the $2N$ term will dominate in the forward pass cost.
- Take GPT-3 as an example: $n_{layer} = 96$, $n_{ctx} = 4096$, $d_{model} = 12288$.

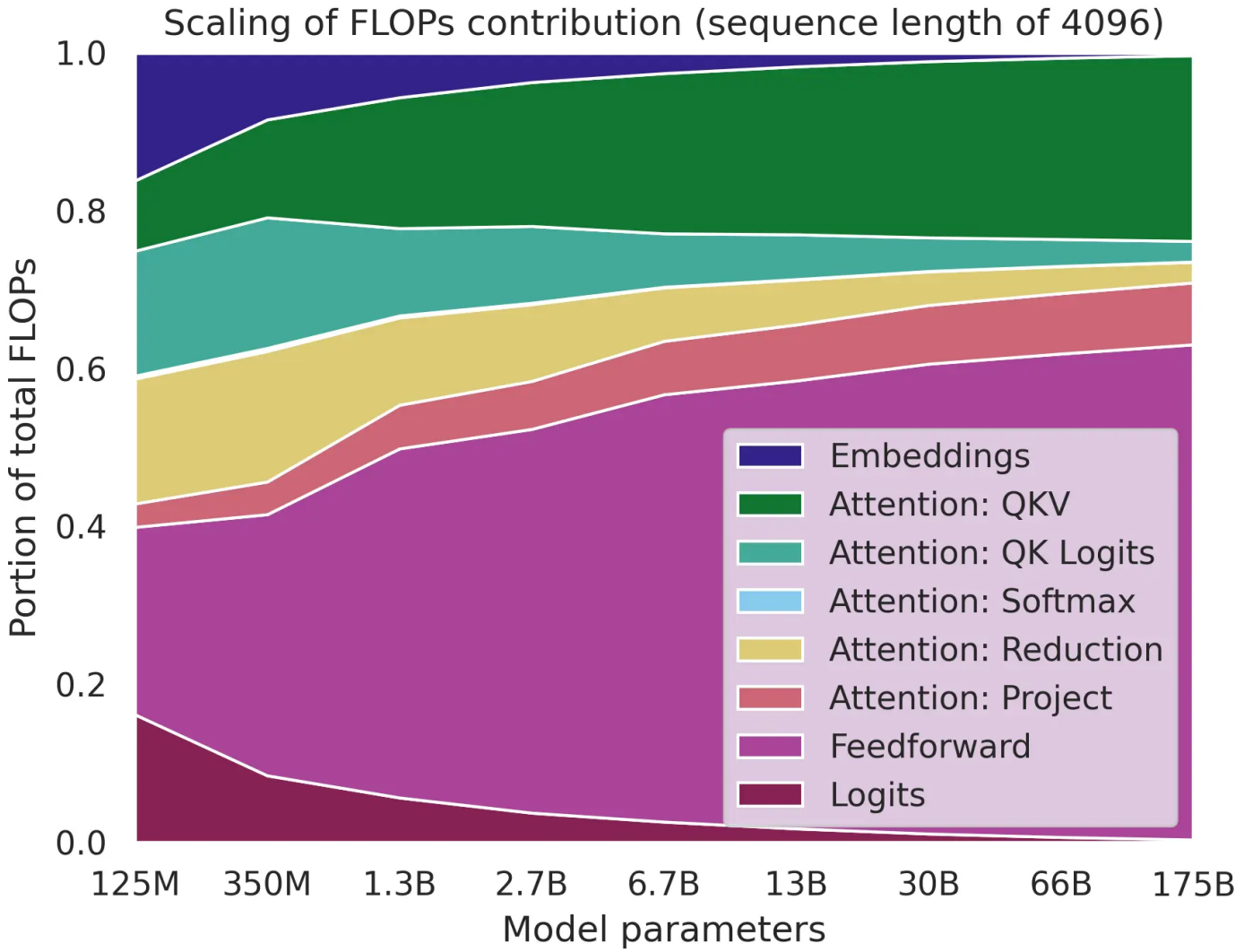
$$N = 12(12288^2)(96) = 174(10^9),$$

$$C_{forward} = 2(174)(10^9) + 2(96)(4096)(12288) = 348(10^9) + 9.7(10^9).$$

- The first term makes up > 97% of the total FLOPs.

[Adam Casson, Transformer FLOPs, 2023]

COST OF FORWARD PASS



[Adam Casson, Transformer FLOPs, 2023]

COST OF TRAINING

- We now know how much it costs to perform a forward pass for a decoder-only transformer.
- But how much does it cost to train?
- Recall that the cost of the transformer is dominated by matrix products.
- Let's examine a single matrix product that happens in the middle of a neural network:

$$Y = XA$$

  - where X is the input activations (from the previous layer),
  - A is the weight matrix,
  - and Y is the output activations (for the next layer).

COST OF TRAINING

$$Y = XA$$

- We want to compute $\frac{\partial L}{\partial A}$, where L is the loss function.
  - By the chain rule:

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial A} = X^{\mathrm{T}}\frac{\partial L}{\partial Y}$$

- But we also need to compute $\frac{\partial L}{\partial X}$, in order to compute the gradients for the previous layer (backpropagation).
  - Again we use the chain rule:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial L}{\partial Y}A^{\mathrm{T}}.$$

- So we need 2 matrix multiplications for each linear layer in the network.

COST OF TRAINING

$$Y = XA$$

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial A} = X^{\mathrm{T}}\frac{\partial L}{\partial Y}$$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial L}{\partial Y}A^{\mathrm{T}}.$$

- Notice we also need to have computed X in the above formula for $\frac{\partial L}{\partial X}$.
- This is why we perform a forward pass before each backward pass.
  - The forward pass computes X for all linear layers.
  - The backward pass computes gradients with respect to A for all linear layers
- The forward pass requires 1 matrix multiplication for each linear layer.
- So computing the gradient requires 3 times as many matrix products.

COST OF TRAINING

- For a transformer, since the forward pass costs $2N$ FLOPs per token, where N is the number of parameters,
- That would imply that each step of training costs $6N$ FLOPs per token.
- How much memory do we need?
  - We need memory to store the model parameters: N
  - We need to store the gradient with respect to each parameter: N
  - The optimizer may also require additional memory:
    - Adam (Kingma and Ba, 2014) stores 2 values per parameter: $2N$
  - The activations require additional memory,
    - But this is not a simple function of N.
    - This grows linearly with respect to batch size!

COST OF TRAINING

- Example memory usage for training GPT-2 Small:

GPT2 Small	Predicted	Actual	Diff
N Parameters	124,373,760	124,373,760	0
Model Memory (bytes)	547,826,688	547,826,688	0
Gradients (bytes)	497,495,040	497,495,040	0
Adam buffers (bytes)[1]	994,990,080	994,990,380	300
CuBLAS workspace (bytes)[2]	17,039,360	17,039,360	0
Gaps (bytes)		6,855,368	-
Inputs / Targets (bytes)	196,608	Not visible	-
Others not visble on segment (bytes)		196,620	-
Total (bytes)[3]	2,057,547,776	2,064,403,456	6,855,680

Batch size = *12*
Total = ~*16.5N*

[Edward Rees, Transformer Memory Arithmetic: Understanding all the Bytes in nanoGPT, 2023]

COST OF TRAINING

- What is more expensive for LLMs? Training or inference?
  - What about for deployed models (e.g., ChatGPT)?
  - How long do they need to be deployed before inference cost exceeds training cost?
- GPT-3 was trained on $300(10^9)$ tokens.
  - So the number of training FLOPs was roughly $6(300)(10^9)N = 1.8(10^{12})N$.
  - How many inference forward passes is this equivalent to?
  - Divide the number of training FLOPs by $2N$:
    - $900(10^9)$ forward passes
  - Suppose each prompt generates 10000 tokens.
    - 90 million prompts would match the cost of training.

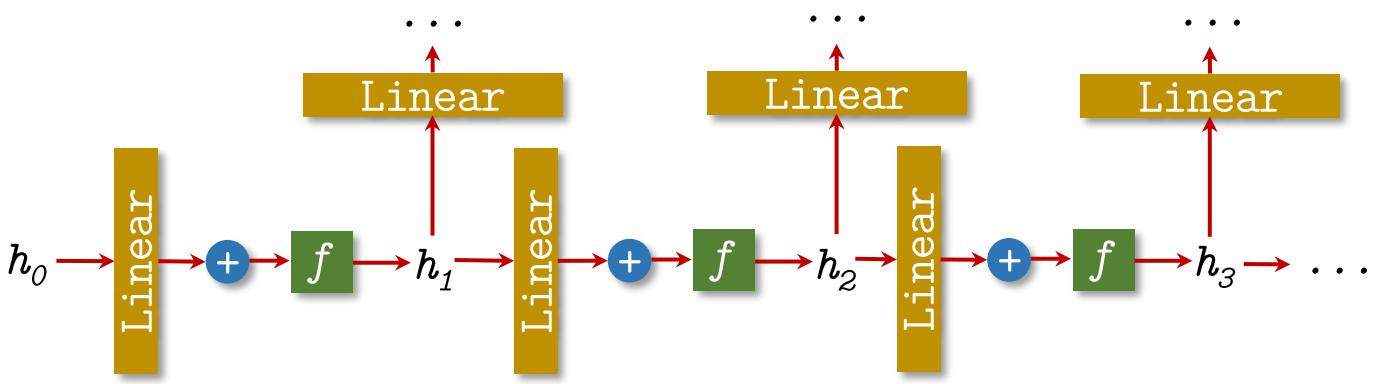
COST OF TRAINING

Disruption and innovation in search don't come for free. The costs to train an LLM, as we detailed here, are high. More importantly, inference costs far exceed training costs when deploying a model at any reasonable scale. In fact, the costs to inference ChatGPT exceed the training costs on a weekly basis. If ChatGPT-like LLMs are deployed into search, that represents a direct transfer of $30 billion of Google's profit into the hands of the picks and shovels of the computing industry.

[Patel and Ahmad, The Inference Cost Of Search Disruption – Large Language Model Cost Analysis, 2023]

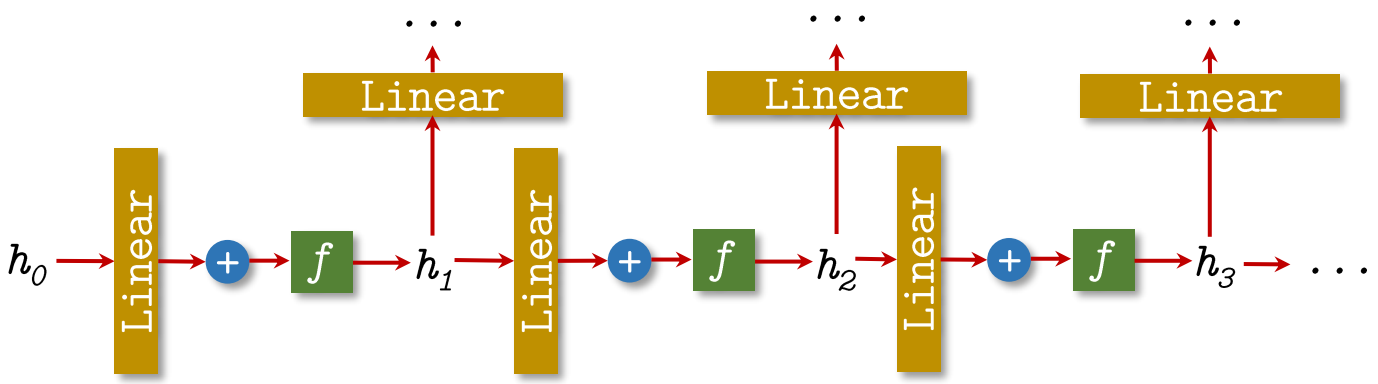
COST OF RNNS

- We can perform the same analysis for RNNs:
  - Assume a “vanilla” RNN with just a single linear layer between hidden states.
  - Each step in the RNN requires computing two linear layers:
    - One to compute the new hidden state from the old one.
    - One to compute the output from the hidden state.



COST OF RNNS

- We can perform the same analysis for RNNs:
  - Each linear layer has a $d_{model} \times d_{model}$ weight matrix and a bias vector with dimension d_{model}.
  - Time: Requires $4(d_{model}^2 + d_{model}) \approx 4d_{model}^2$ FLOPs.
  - Memory: Requires $2(d_{model}^2 + d_{model}) \approx 2d_{model}^2$ parameters.



COST OF RNNS

- We can perform the same analysis for RNNs:
  - So if we have an input sequence length of n_{ctx} and n_{layer} layers,
  - The total number of parameters is:

    $N = 2n_{layer}{d_{model}}^2$, (since parameters are shared)
  - And the total number of FLOPs per forward pass is:

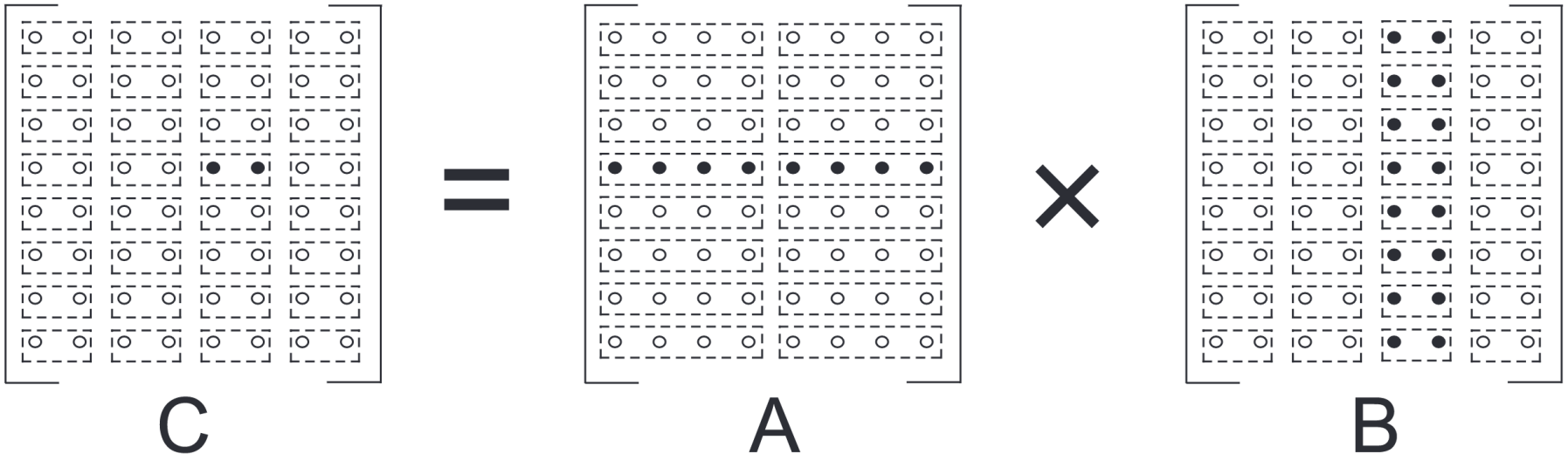
    $C_{forward} = 4n_{ctx}n_{layer}{d_{model}}^2 = 2n_{ctx}N.$
- Similar to the transformer, most of the computation is dominated by matrix multiplications.
- So the cost of each training step is $6n_{ctx}N$.

TRANSFORMERS VS RNNS

- RNN: $C_{forward} = 4n_{ctx}n_{layer}d_{model}^2$,
- Transformer: $N = 12n_{layer}d_{model}^2$ per token,
  - So if we have n_{ctx} tokens in the input, then $C_{forward} = 24n_{ctx}n_{layer}d_{model}^2$.
- The two models have the same computational cost, asymptotically.
- So why are transformers better?
  - Their operations are much more parallelizable.
  - For an RNN, we have to wait for the previous hidden state before we can compute the next one.

PARALLELIZATION

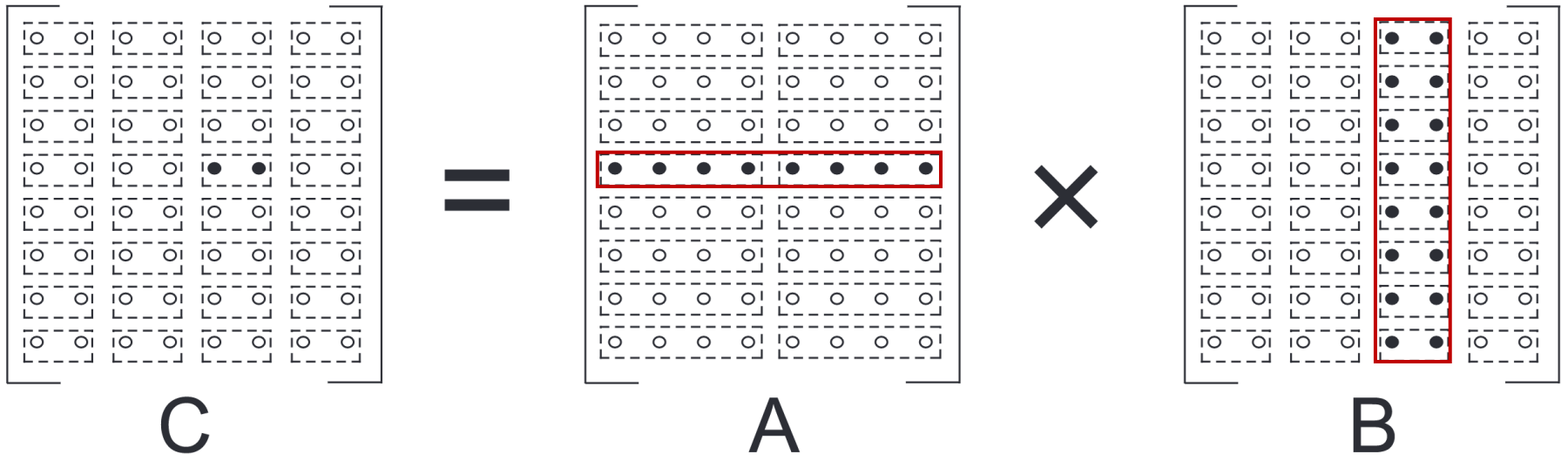
- But how do we parallelize matrix multiplication?
- Suppose we have many available threads.
- And we want to compute: $C = AB$.



[Li et al., GPU Matrix Multiplication, Multi- and Many-Core Technologies: Architectures, Programming, Algorithms and Applications, 2013]

PARALLELIZATION

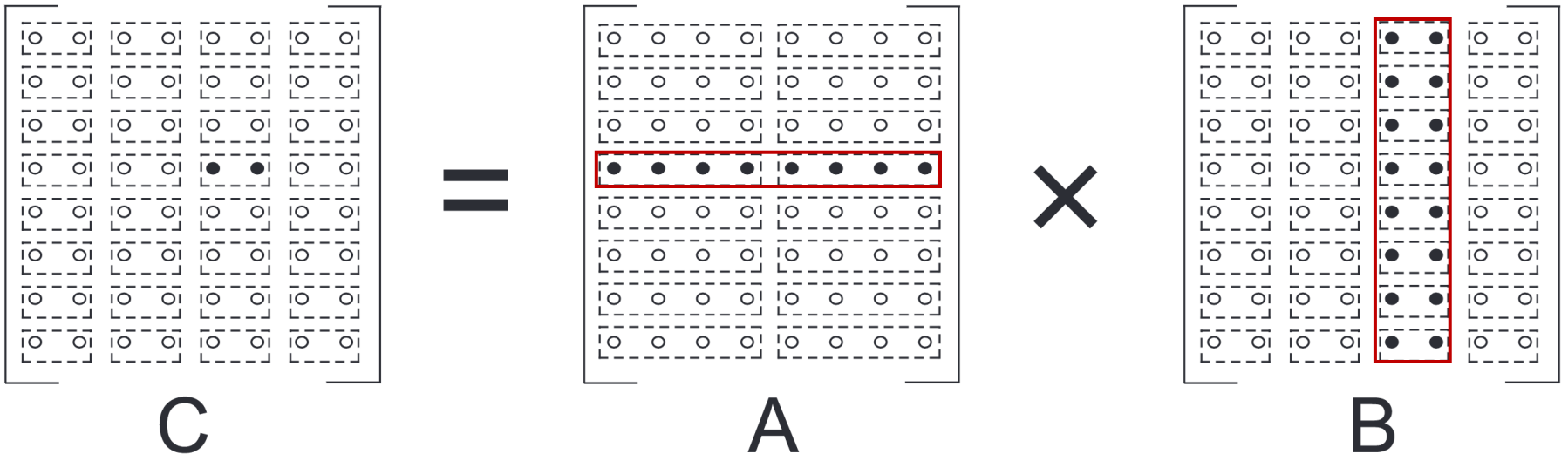
- We can tile the matrices into submatrices.
- For example, we can assign the task of computing the highlighted `1 x 2` submatrix in C to one thread.
  - It multiplies the `1 x 8` submatrix in A with the `8 x 2` submatrix in B.



[Li et al., GPU Matrix Multiplication, Multi- and Many-Core Technologies: Architectures, Programming, Algorithms and Applications, 2013]

PARALLELIZATION

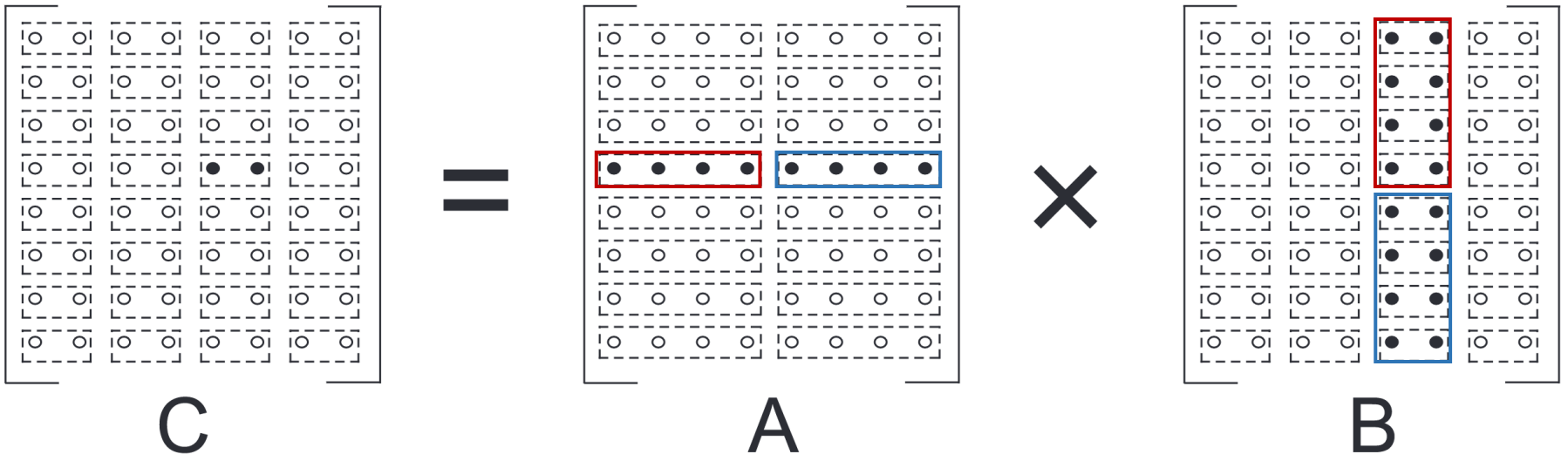
- We can tile the matrices into submatrices.
- With this approach, if $A \in \mathbb{R}^{n \times k}$ and $B \in \mathbb{R}^{k \times m}$, and we have nm threads,
- The matrix multiplication can be completed in $2k$ operations.



[Li et al., GPU Matrix Multiplication, Multi- and Many-Core Technologies: Architectures, Programming, Algorithms and Applications, 2013]

PARALLELIZATION

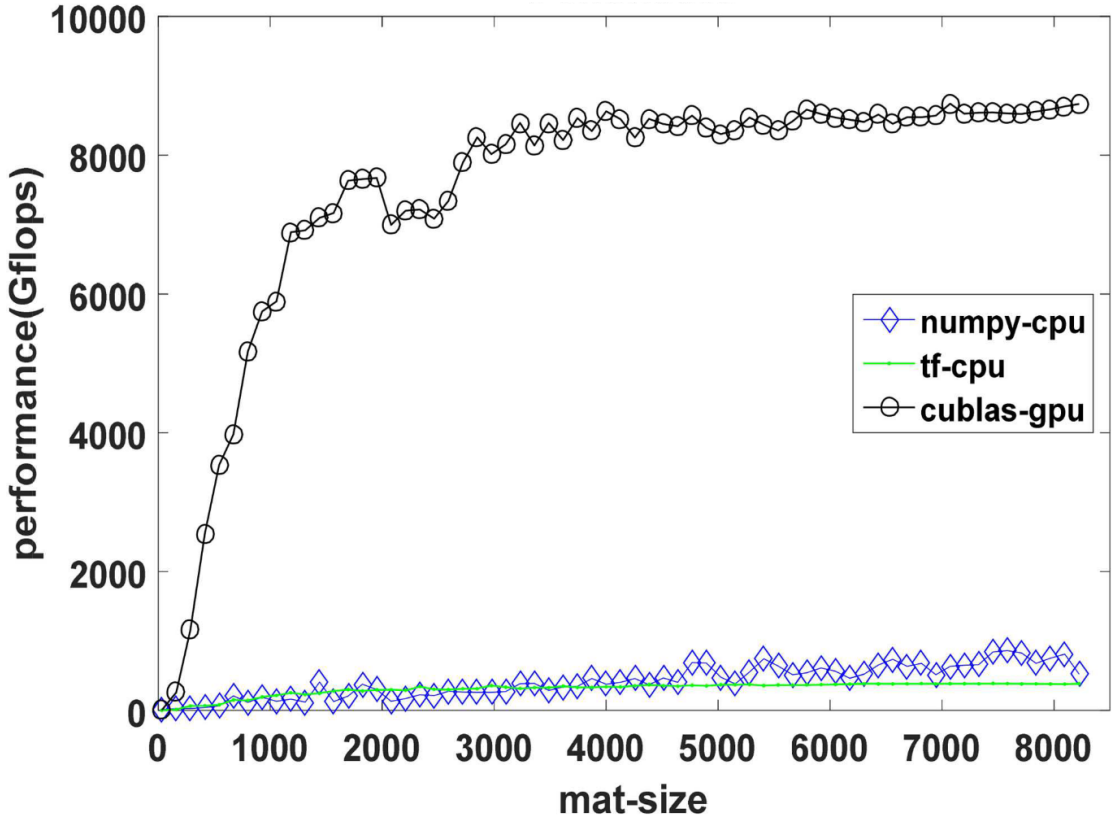
- We can break the matrices down even further:
  - Have one thread multiply the left portion of the highlighted submatrix in A (1 x 4) with the top portion of the highlighted submatrix in B (4 x 2).
  - Have a different thread multiply the right submatrix in A with the bottom portion in B.
  - Then the two resulting 1 x 2 matrices would be summed to produce the result in C.



[Li et al., GPU Matrix Multiplication, Multi- and Many-Core Technologies: Architectures, Programming, Algorithms and Applications, 2013]

GPUS VS CPUS

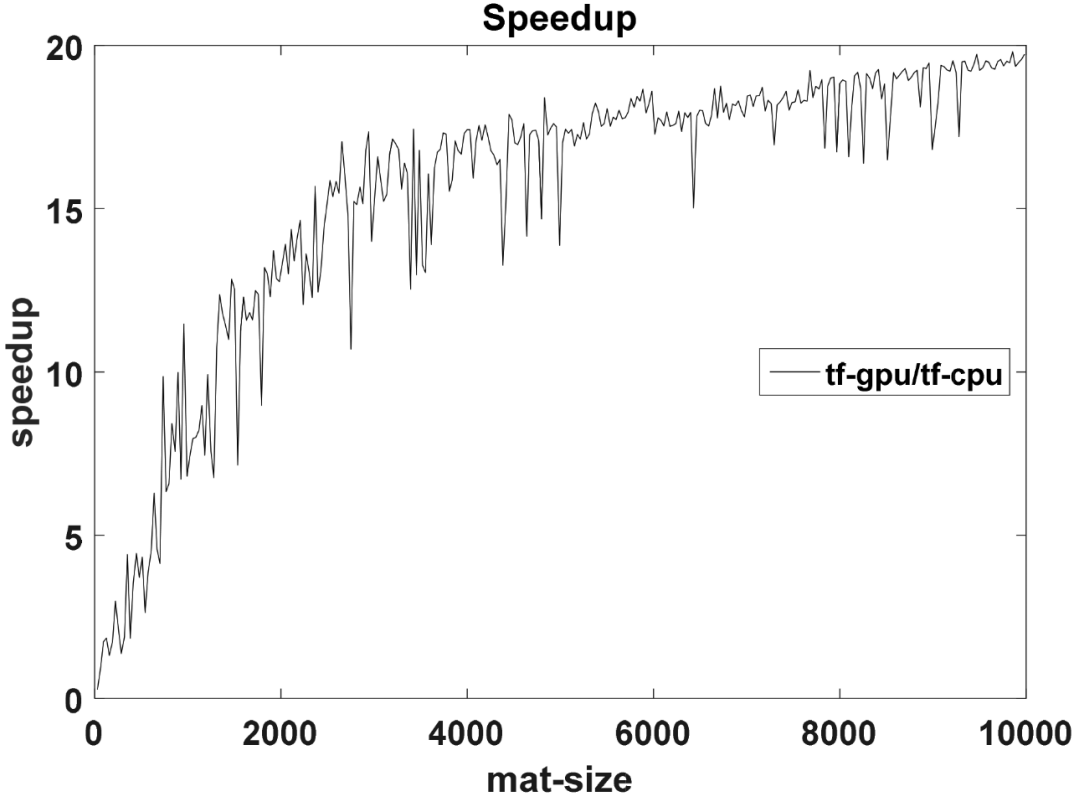
- GPUs are much better than CPUs at parallel processing.



[Huang et al., GPU computing performance analysis on matrix multiplication, 2019]

GPUS VS CPUS

- GPUs are much better than CPUs at parallel processing.



[Huang et al., GPU computing performance analysis on matrix multiplication, 2019]

GPU PROGRAMMING

- In order to maximize the benefit from using GPUs, we need to keep all our matrices in GPU memory.
- So for ML applications, that requires that all operations required by the model forward/backward pass are implemented on the GPU.
- It is (relatively) expensive to transfer data to/from GPU memory and system memory.
- If you develop a new model that uses a novel operation, and you want to be able to run your model on the GPU,
  - You will need to implement that operation in the GPU.
  - How can we do this?

GPU PROGRAMMING

- GPUs were initially designed to accelerate 3D rendering.
- Initially, they were not programmable.
- In 2001, the first GPUs were released that supported programmable shaders.
  - Developers could write shaders to better customize the rendering process (e.g., how to color each pixel, calculate lighting, etc).
- In 2007, general-purpose GPU programming languages were released to allow GPUs to be used for non-graphics applications.
- There are a number of GPU programming languages that can be used to implement functions on the GPU.
  - CUDA, OpenCL, etc...
- A GPU program that implements a specific routine is called a kernel.

GPU PROGRAMMING

- Example of a matrix-vector product kernel in OpenCL:

```
// Multiplies A*x, leaving the result in y.
// A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].
__kernel void matvec(__global const float *A, __global const float *x,
                     uint ncols, __global float *y)
{
    size_t i = get_global_id(0);              // Global id, used as the row index
    __global float const *a = &A[i*ncols];    // Pointer to the i'th row
    float sum = 0.f;                          // Accumulator for dot product
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```

[https://en.wikipedia.org/wiki/OpenCL]

GPU PROGRAMMING

- Example of element-wise product kernel in CUDA:

```
import pycuda.compiler as comp
import pycuda.driver as drv
import numpy
import pycuda.autoinit

mod = comp.SourceModule(
    """
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
"""
)

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(drv.Out(dest), drv.In(a), drv.In(b), block=(400, 1, 1))

print(dest - a * b)
```

[https://en.wikipedia.org/wiki/CUDA]

GPU PROGRAMMING

- What would a matrix multiplication kernel look like?

```
__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                            const float *B, float beta, float *C) {
  // compute position in C that this thread is responsible for
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  // `if` condition is necessary for when M or N aren't multiples of 32.
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

[Simon Boehm, How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog, 2022]

GPU PROGRAMMING

- What would a matrix multiplication kernel look like?
- This simple implementation runs at 309.0 GFLOPs/s on an A6000 GPU.
- Nvidia's proprietary implementation (in the cuBLAS library) performs the same operation at 23249.6 GFLOPs/s!
- The naïve implementation is 1.3% as fast as the proprietary one.
- What accounts for this huge performance difference?
  - Optimizations!

[Simon Boehm, How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog, 2022]

GPU PROGRAMMING

- Matrix multiplication optimizations:

Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%

29 lines of code

187 lines of code

[Simon Boehm, How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog, 2022]

GPU PROGRAMMING

- Linear algebra optimizations can provide significant speedups.
- However, they are tedious to implement.
- Some optimizations must be aware of the hardware architectural details.
  - Cache structure, scheduling, number and types of cores, etc.
- Optimizations are frequently updated for new generations of hardware.

GPU PROGRAMMING

- Thanks to modern ML libraries such as TensorFlow and PyTorch, much of this complexity is abstracted away from us.

```
>>> mat1 = torch.randn(2, 3)
>>> mat2 = torch.randn(3, 3)
>>> torch.mm(mat1, mat2)
tensor([[ 0.4851,  0.5037, -0.3633],
        [-0.0760, -3.6705,  2.4784]])
```

- These libraries will call the GPU kernels to execute this operation quickly.
- These libraries also implement automatic differentiation.
  - We only need to write code for the forward pass.
  - TensorFlow/PyTorch will automatically compute the gradient (autograd).

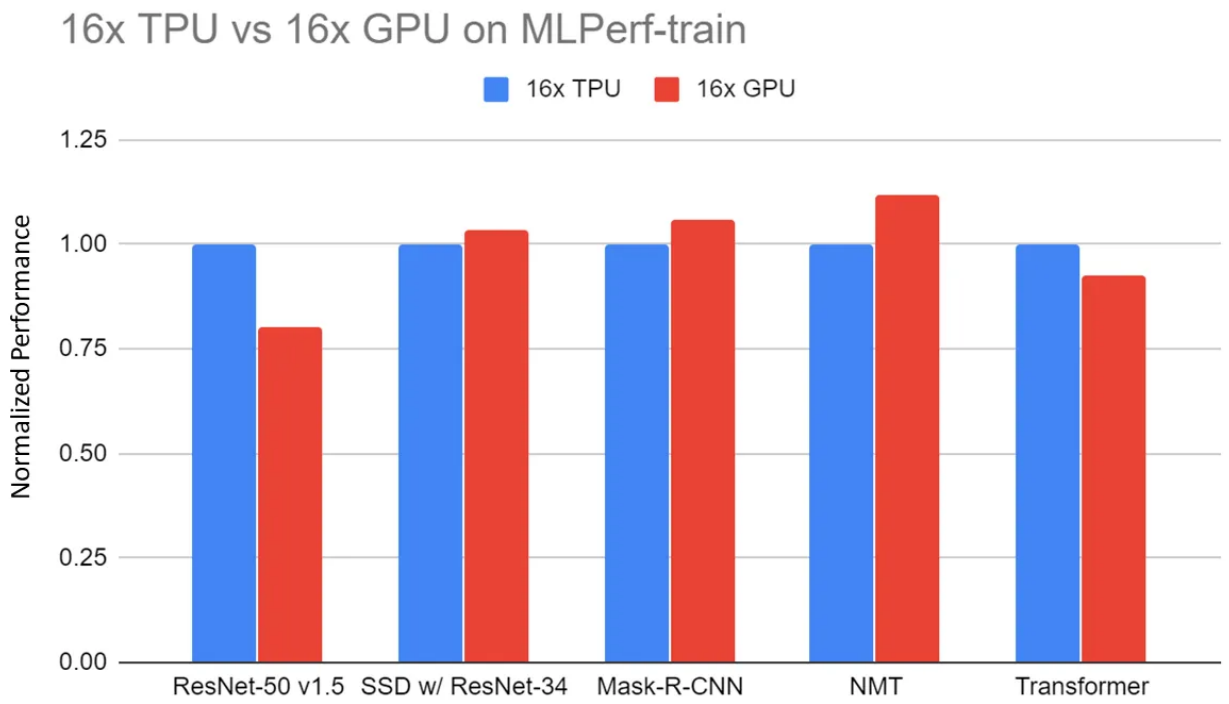
GPU PROGRAMMING

- But if you develop a new model that requires a new operation that is not provided by existing CUDA or OpenCL libraries, then you will need to write a kernel that implements this operation.
- You may also need to write code that automatically differentiates this operation so that TensorFlow/PyTorch can correctly do so when running the backward pass.

MORE SPECIALIZED HARDWARE

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

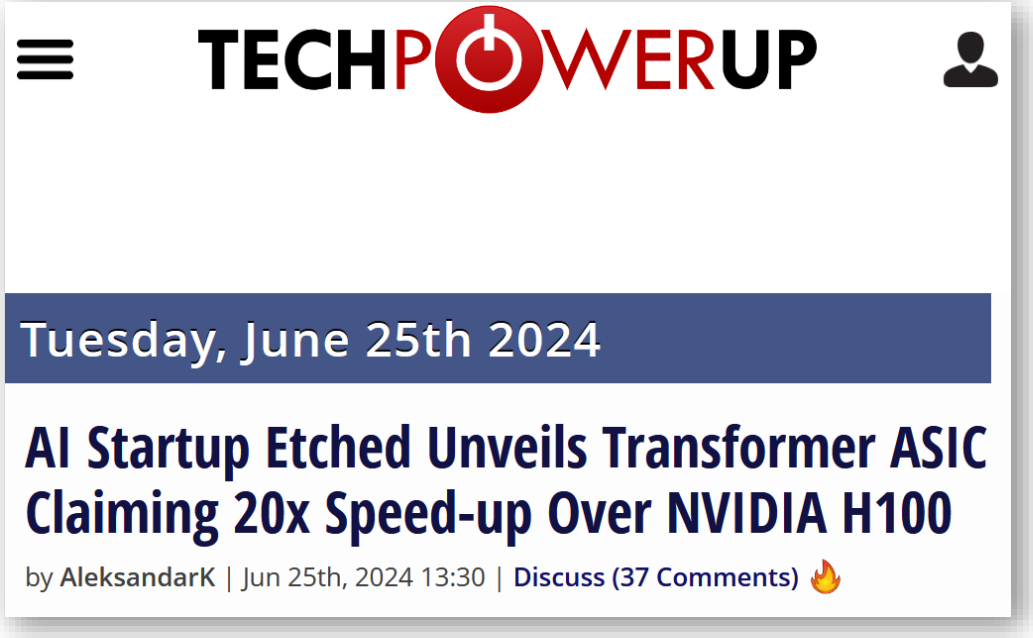
Performance is slightly better.



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

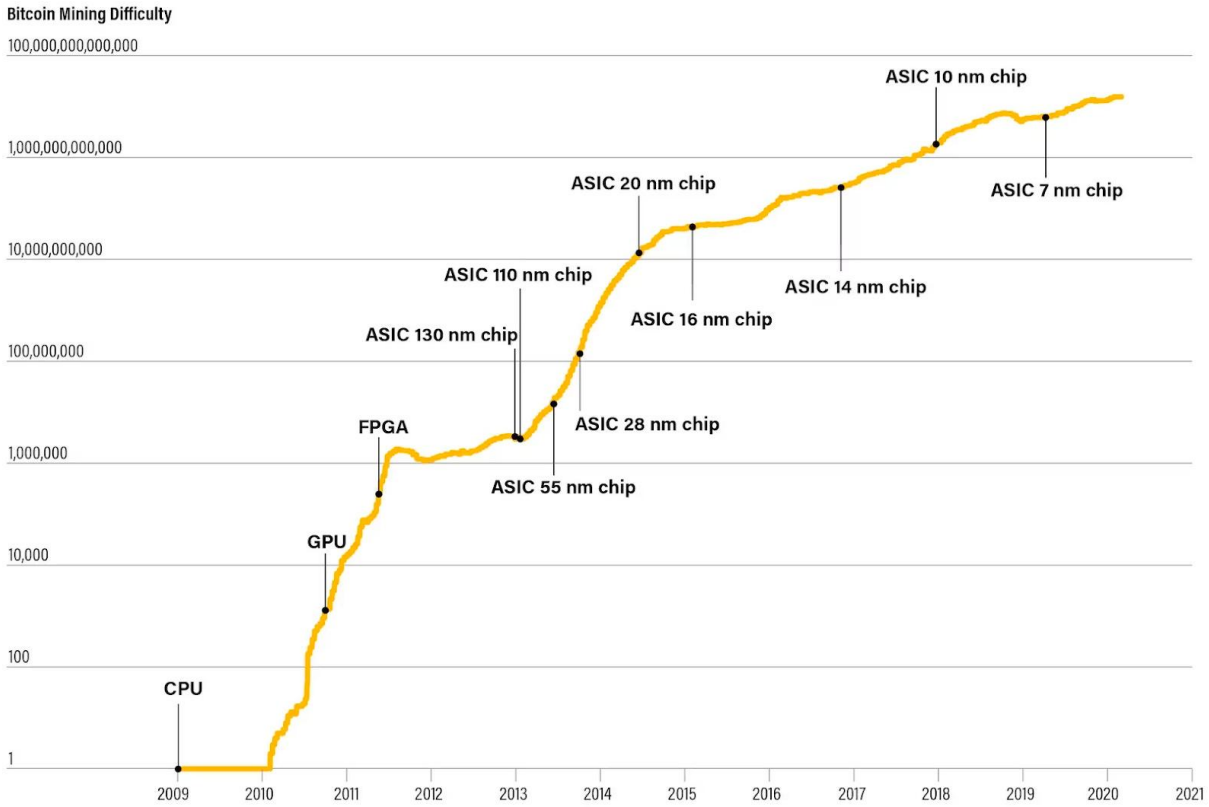
MORE SPECIALIZED HARDWARE

- 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]

NEXT TIME: 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.
- But what if the models/matrices can't fit in the memory of one GPU?
- 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)

QUESTIONS?