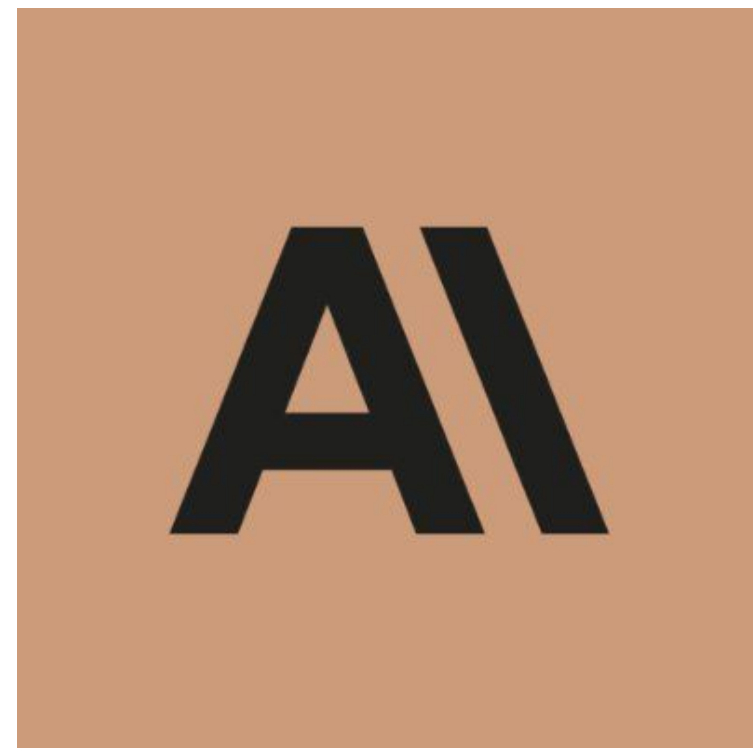


Building Machine Learning  
Systems For A Trillion Trillion  
Floating Point Operations  
(and why you should care)

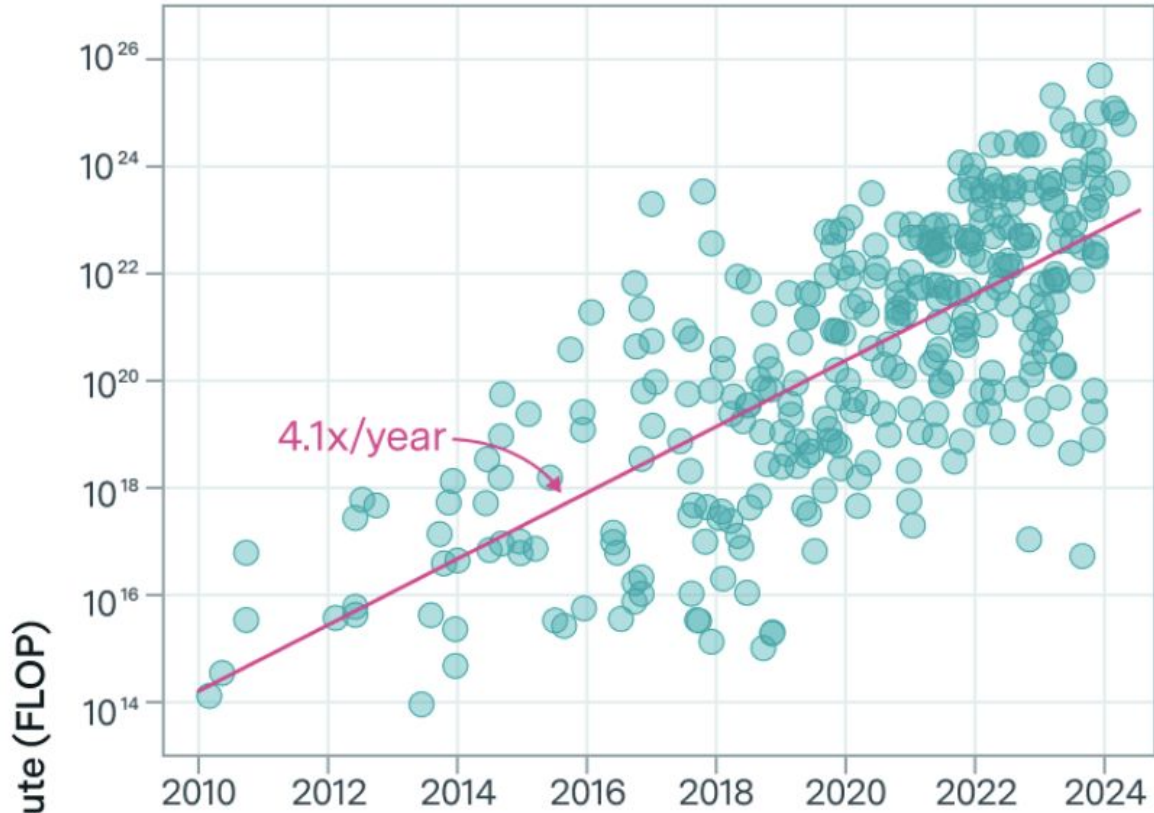
**Horace He**  
**Pytorch Compilers**

# We Live in Unprecedented Times

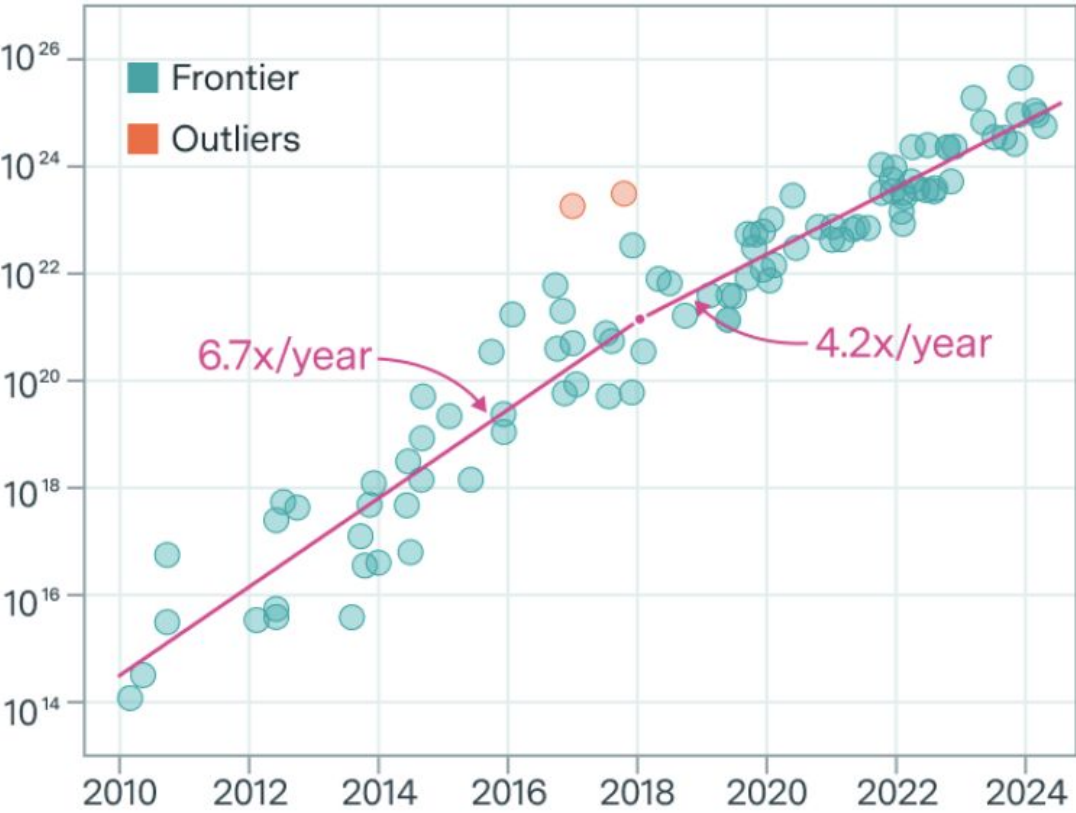


# Floating Point Operations

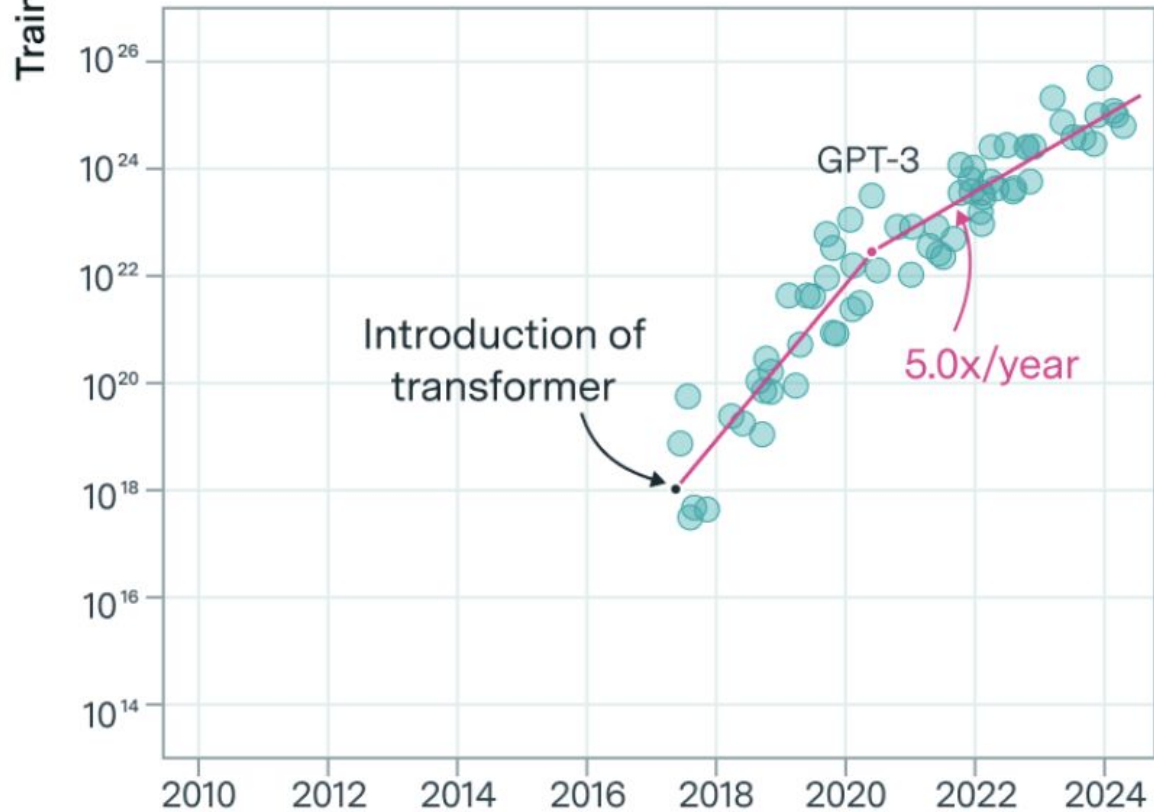
Notable models



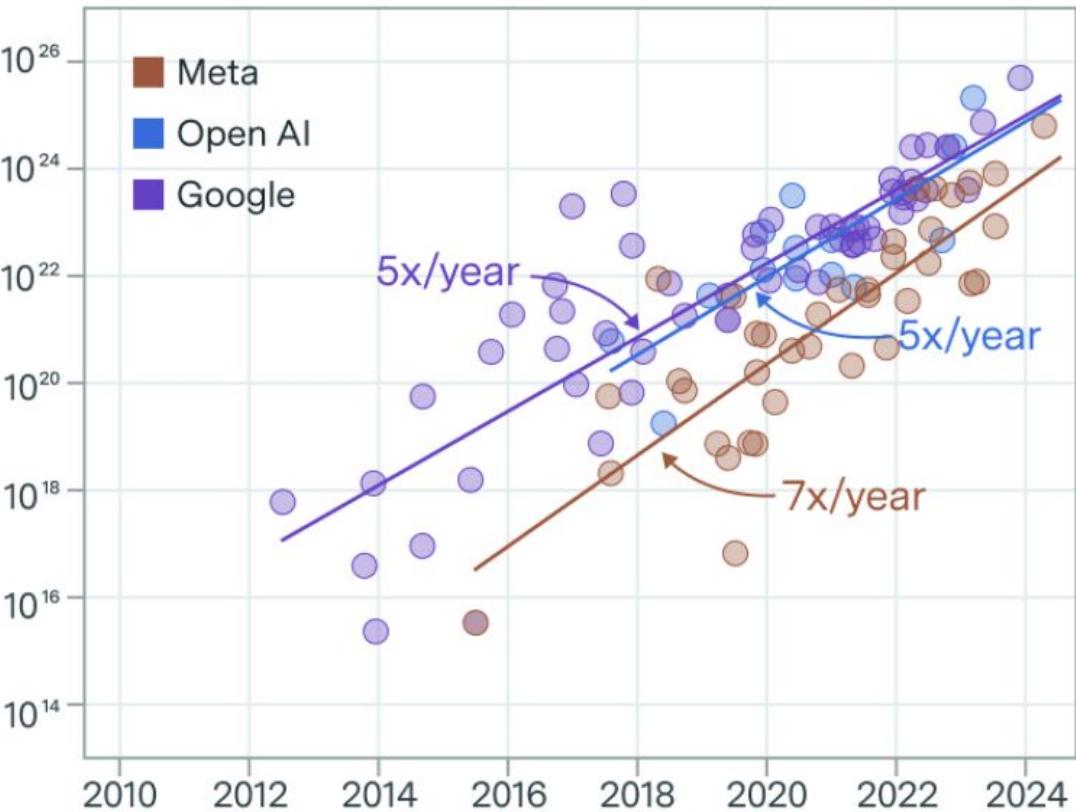
Frontier models



Frontier LLMs



Select companies



Publication date

# Stock Prices

Market Summary > NVIDIA Corp

## 145.61 USD

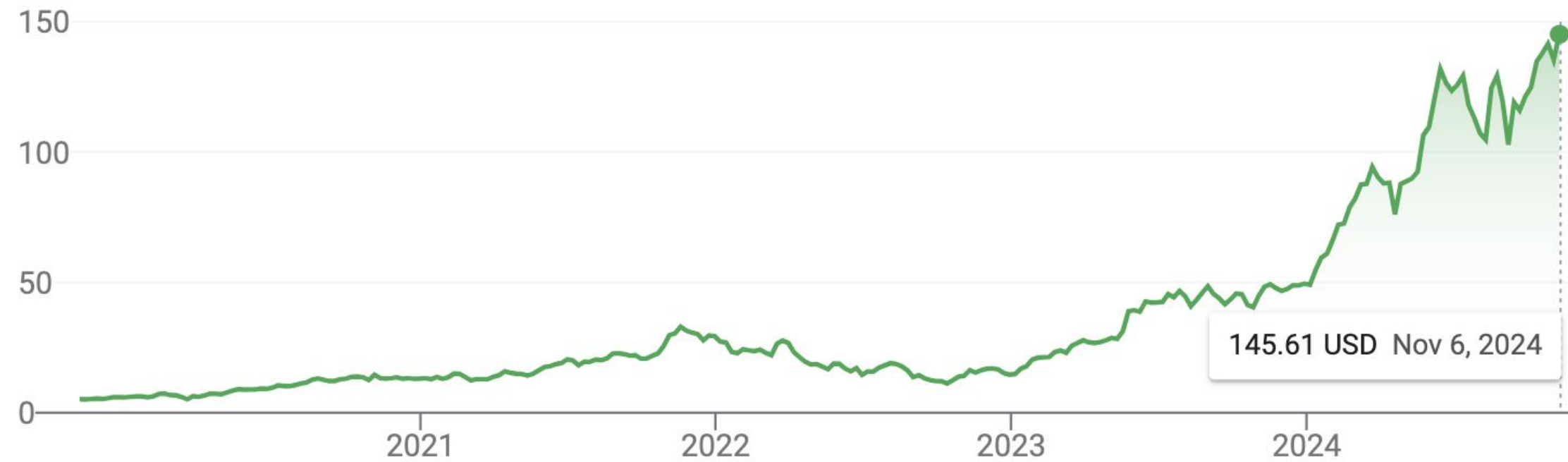
+ Follow

+140.42 (2,705.59%) ↑ past 5 years

Closed: Nov 6, 7:59 PM EST • Disclaimer

After hours 145.30 -0.31 (0.21%)

1D | 5D | 1M | 6M | YTD | 1Y | **5Y** | Max



Open	142.96	Mkt cap	3.57T	CDP score	B
High	146.49	P/E ratio	68.39	52-wk high	146.49
Low	141.96	Div yield	0.027%	52-wk low	44.90

# Microsoft, OpenAI plan \$100 billion data-center project, media report says

By Reuters

March 29, 2024 5:14 PM EDT · Updated 7 months ago



 Fortune


## Google will help build seven nuclear reactors to power its AI systems

Google is adding nuclear plants to its seemingly ever-growing portfolio. The company has partnered with Kairos Power to back the construction of seven small...



BUSINESS

# The AI boom may give Three Mile Island a new life supplying power to Microsoft's data centers

 Data Center Dynamics

## Elon Musk claims 300,000 B200 GPU supercomputer for xAI by next summer; the usual caveats apply

Elon Musk claims that artificial intelligence startup xAI will deploy a 300,000 Nvidia Blackwell B200 GPU data center by next summer.

Jun 3, 2024





Reuters

## Exclusive: OpenAI co-founder Sutskever's new safety-focused AI startup SSI raises \$1 billion

SAN FRANCISCO/NEW YORK, Sept 4 - Safe Superintelligence (SSI), newly co-founded by OpenAI's former chief scientist Ilya Sutskever,...

Sep 4, 2024



CNBC

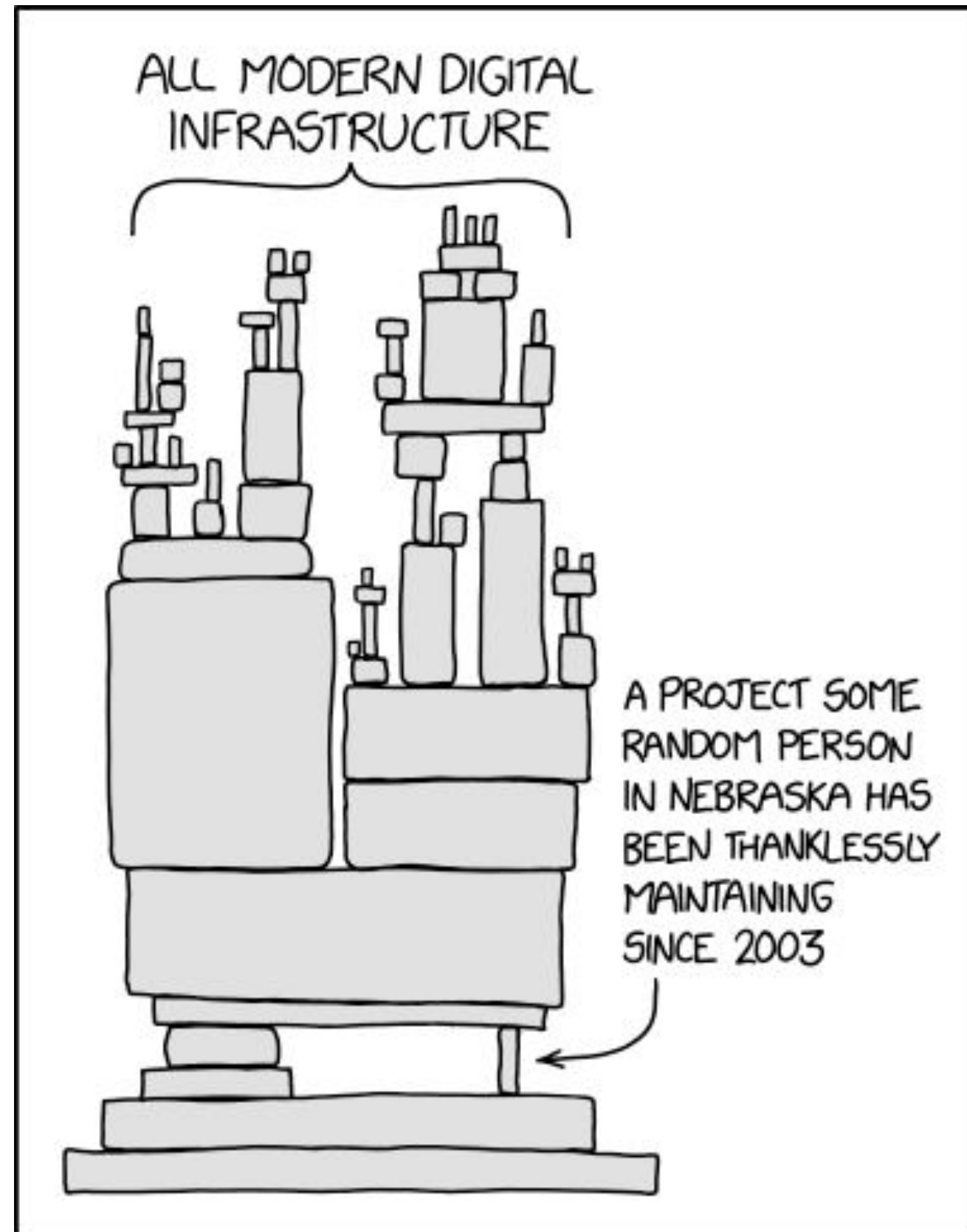
## Microsoft-backed Mistral AI raises \$645 million at a \$6 billion valuation

Microsoft-backed Mistral AI raises \$645 million at a \$6 billion valuation ... French founder of artificial intelligence startup Mistral AI, Arthur...

Jun 12, 2024



# How do we actually build anything with these systems?



Prompting/End User Applications

Machine Learning Models

ML Software

ML Hardware

Semiconductors

Physics



Prompting/End User Applications

Machine Learning Models

ML Software

ML Hardware

Semiconductors

Physics

I (mostly)  
work here



Prompting/End User Applications

Machine Learning Models

ML Software

ML Hardware

Semiconductors

Physics

Many of you  
are here



# The modern world is all about layers of abstractions

An abstraction is an attempt to hide details of your system from you, allowing you to avoid worrying about those details.

- A transistor abstracts away the “physical world” into the digital.
- A GPU abstracts away all of the transistors into a single hardware unit.
- PyTorch abstracts away the details of how a matmul gets executed on a GPU into operations on tensors.
- LLMs abstract away the tensor operations into an API call that takes in text and returns text
- Agentic Systems abstract away the LLM API calls into “doing something”

Prompting/End User Applications

Machine Learning Models

ML Software

ML Hardware

Semiconductors

Physics

ML  
Systems



# Why should you care about ML systems?

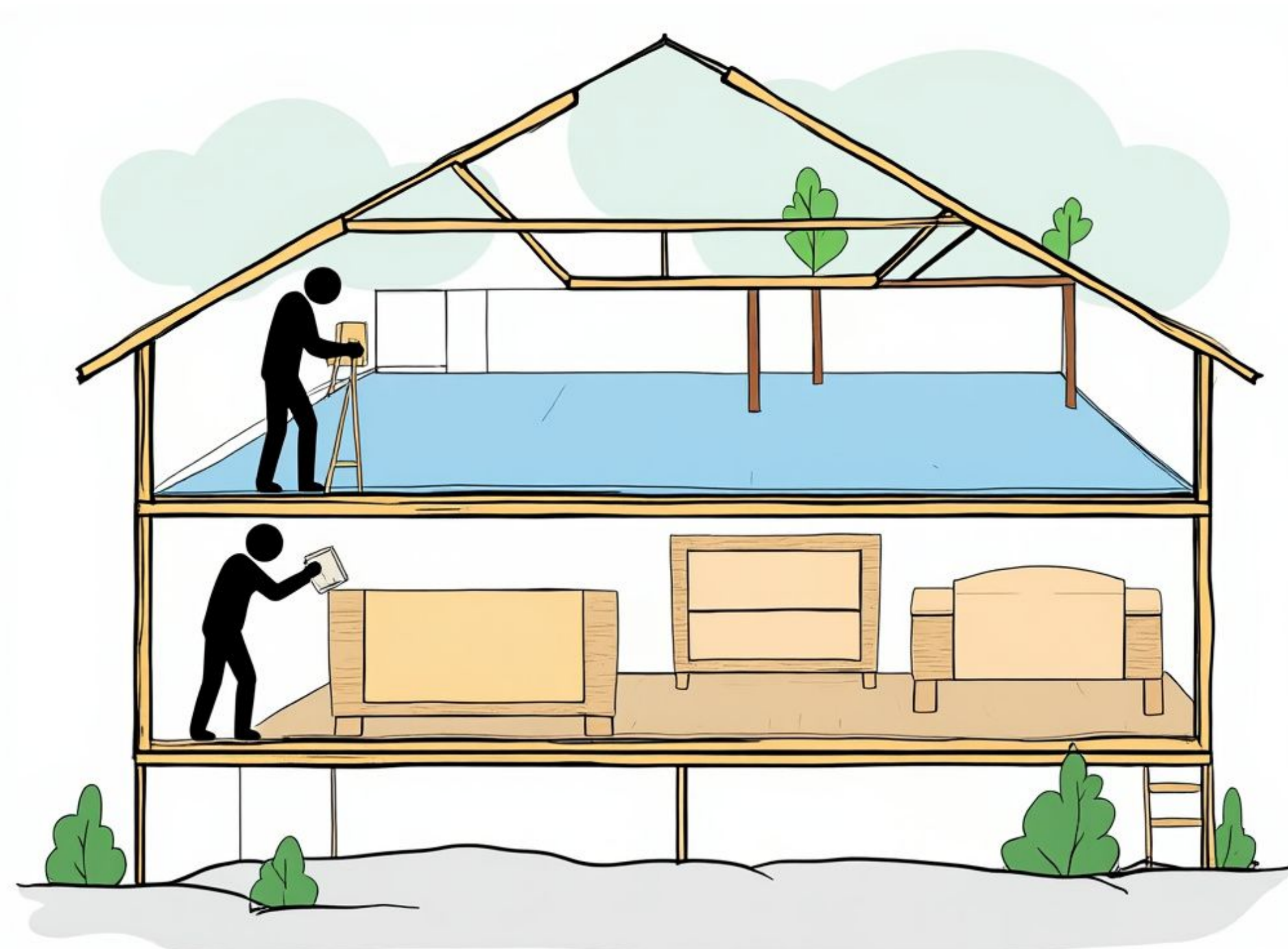
**Law of Leaky Abstractions: All non-trivial abstractions, to some degree, are leaky.**

Understanding the layers

below you allows you to

1. Know when your abstractions are limiting you.

2. Know where they're moving.



# Understand when your abstractions are limiting you

## Convolutional Neural Networks for Object Classification in CUDA

Alex Krizhevsky (kriz@cs.toronto.edu)

April 16, 2009

# And when they're not...



**Sasha Rush**

@srush\_nlp



Btw, one reason this is underused is because log-space is still hard and slow in 2021.

If someone (@NVIDIAAI?), would implement a fast log-space MM [(A[..., None] + B[None]).logsumexp(-2)] it would be amazing. Been failing to do this for years, Cutlass is hard 😞

# You should understand how your abstractions are being used too!

*[Submitted on 6 Nov 2019]*

## **Fast Transformer Decoding: One Write-Head is All You Need**

[Noam Shazeer](#)

Multi-head attention layers, as used in the Transformer neural sequence model, are a powerful alternative to RNNs for moving information across and between sequences. While training these layers is generally fast and simple, due to parallelizability across the length of the sequence, incremental inference (where such parallelization is impossible) is often slow, due to the memory-bandwidth cost of repeatedly loading the large "keys" and "values" tensors. We propose a variant called multi-query attention, where the keys and values are shared across all of the different attention "heads", greatly reducing the size of these tensors and hence the memory bandwidth requirements of incremental decoding. We verify experimentally that the resulting models can indeed be much faster to decode, and incur only minor quality degradation from the baseline.



# Themes of this Lecture

1. It's worth venturing out from your layer of the stack.
2. How should you “interact” with other layers of the stack, both above and below you?
3. I would like to try and explain as much of the ML systems stack as I can.

Prompting/End User Applications

Machine Learning Models

ML Software

ML Hardware

Semiconductors

Physics



Prompting/End User Applications

Machine Learning Models

ML Software

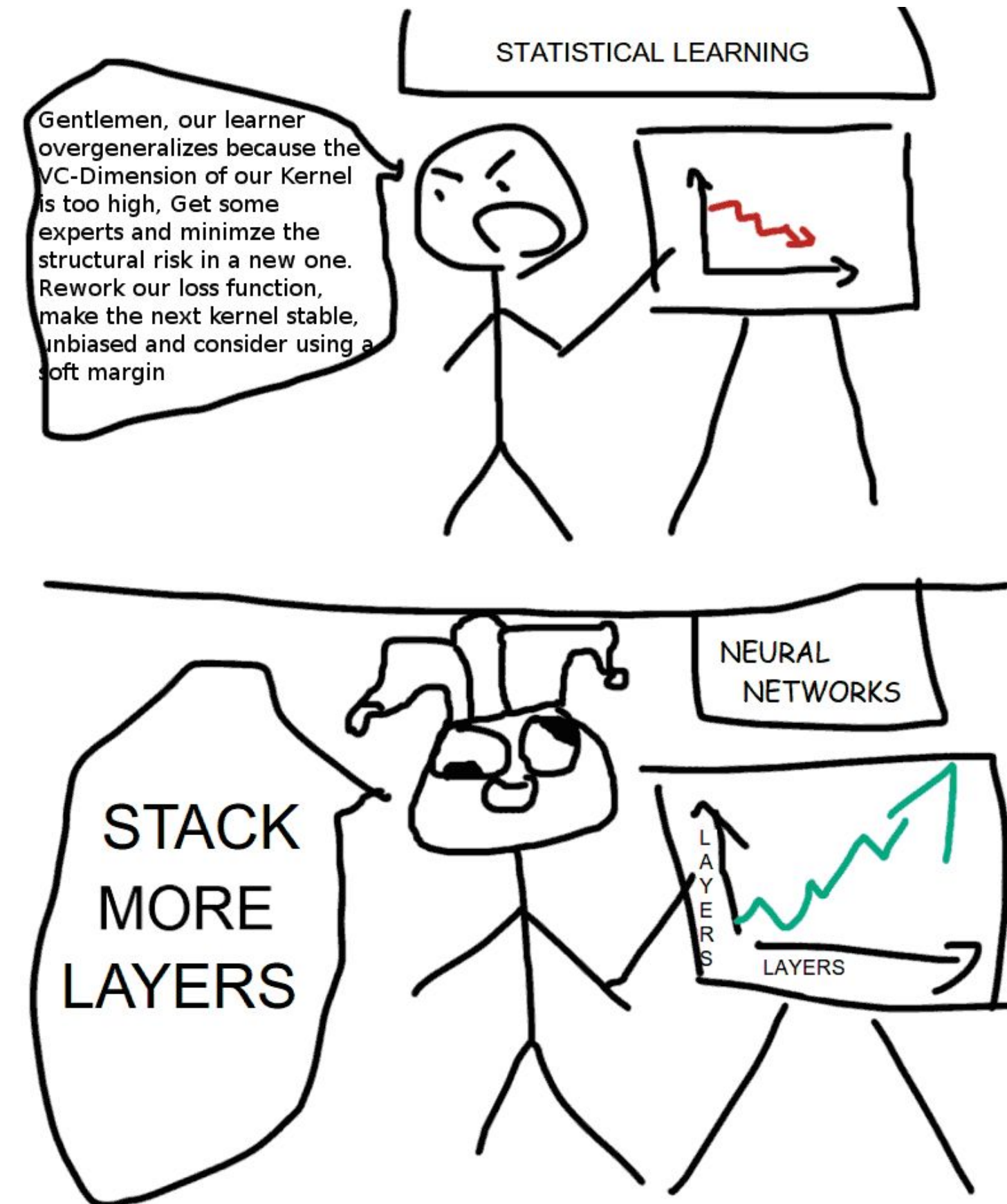
ML Hardware

Semiconductors

Physics

# The Bitter Lesson (Rich Sutton)

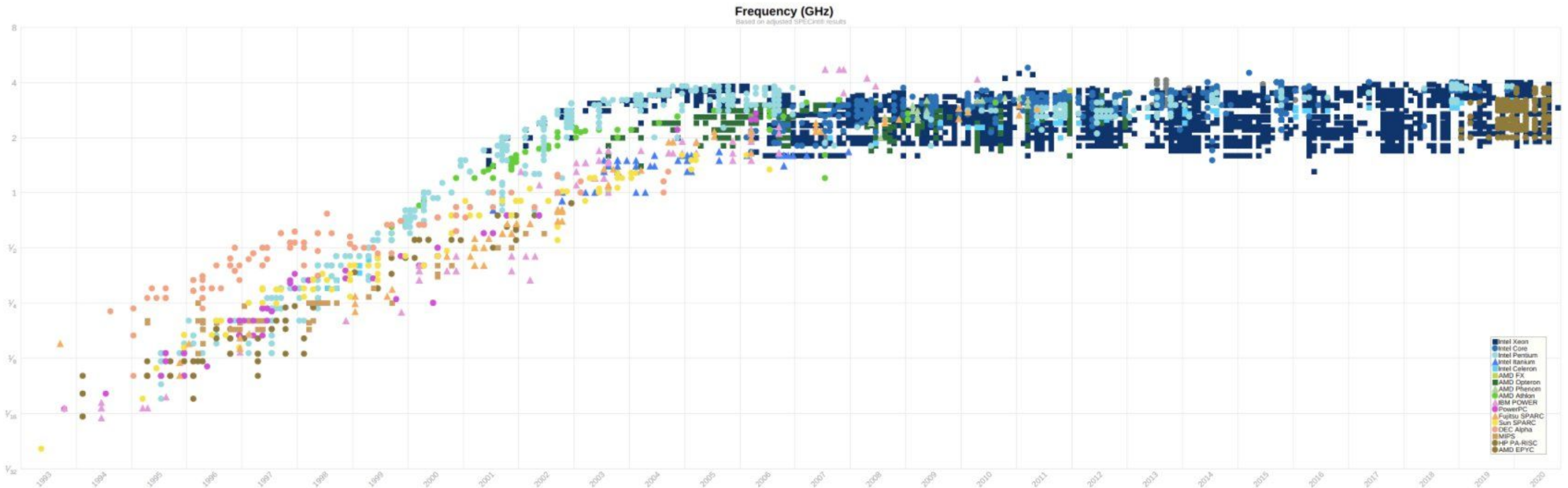
“The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin.”





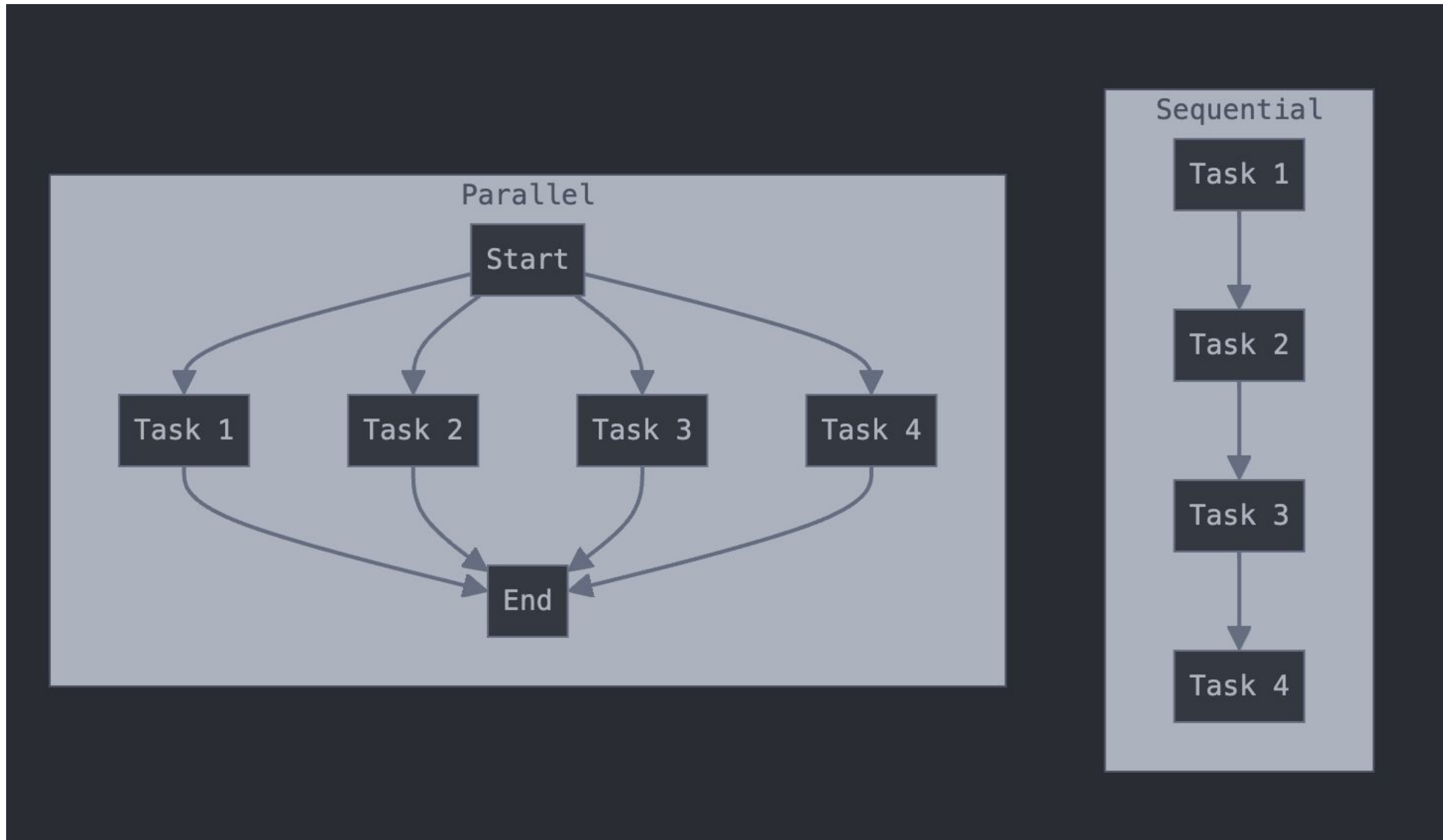
# Death of Dennard Scaling

Dennard Scaling => When transistors get smaller, clock frequency doubles.  
Death Caused by Power Leakage! (Quantum Tunneling)



Transistors got exponentially faster until the moment they didn't.

# Parallel vs. Sequential





# Latency Lags Throughput

End of Dennard Scaling

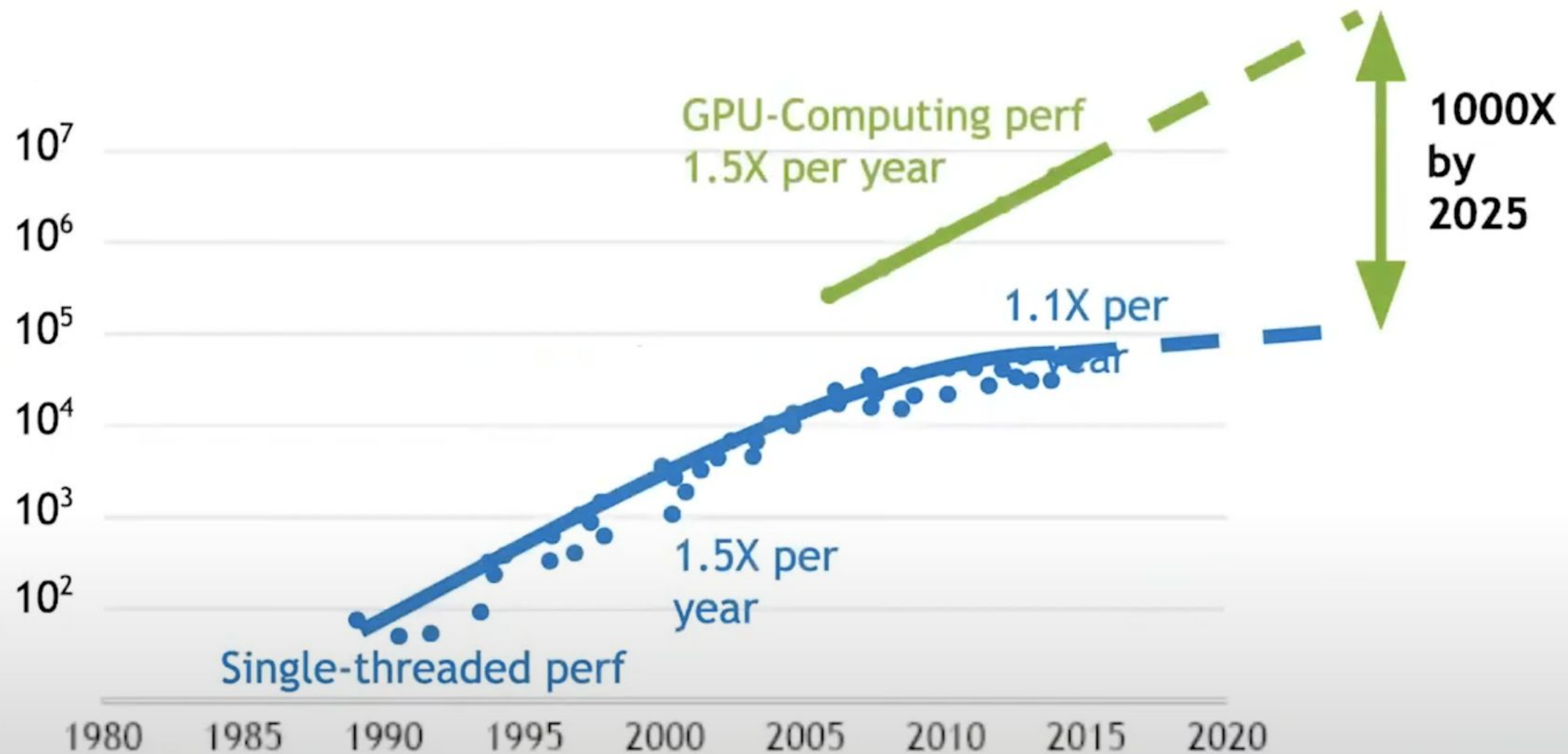
Latency limited by distance

Latency helps bandwidth, but not vice versa.

Increasing bandwidth can hurt latency

<https://claude.site/artifacts/5d014c78-1f9a-44e9-9ba9-0b66a1a00268>

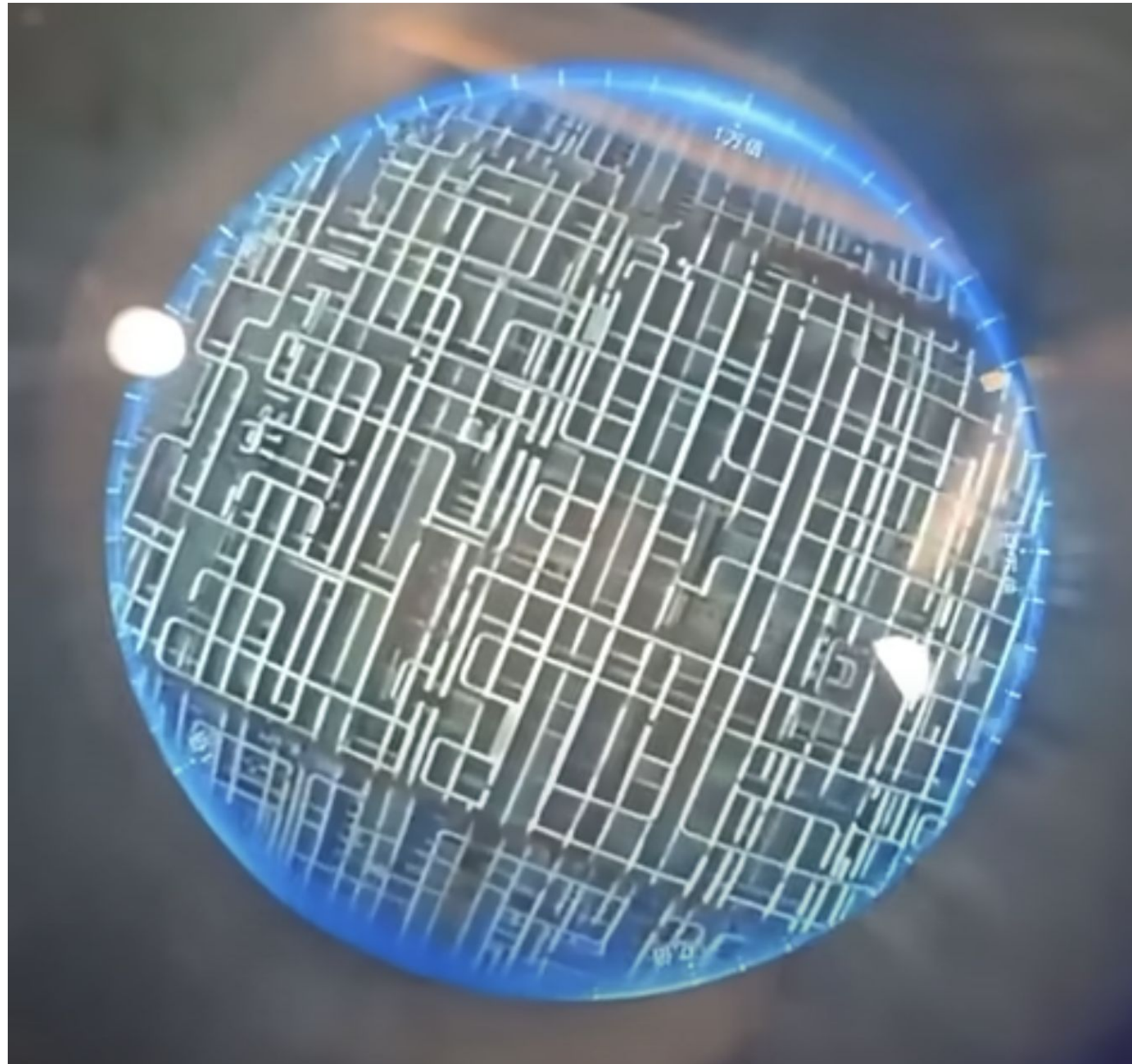
# Death of Dennard Scaling => Massive Parallelism



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

# Semiconductor Manufacturing is Pretty Insane

<https://www.youtube.com/watch?v=xZIZ3LWyhvc> (watch)



# Recommend: Branch Education “How are Microchips Made?”

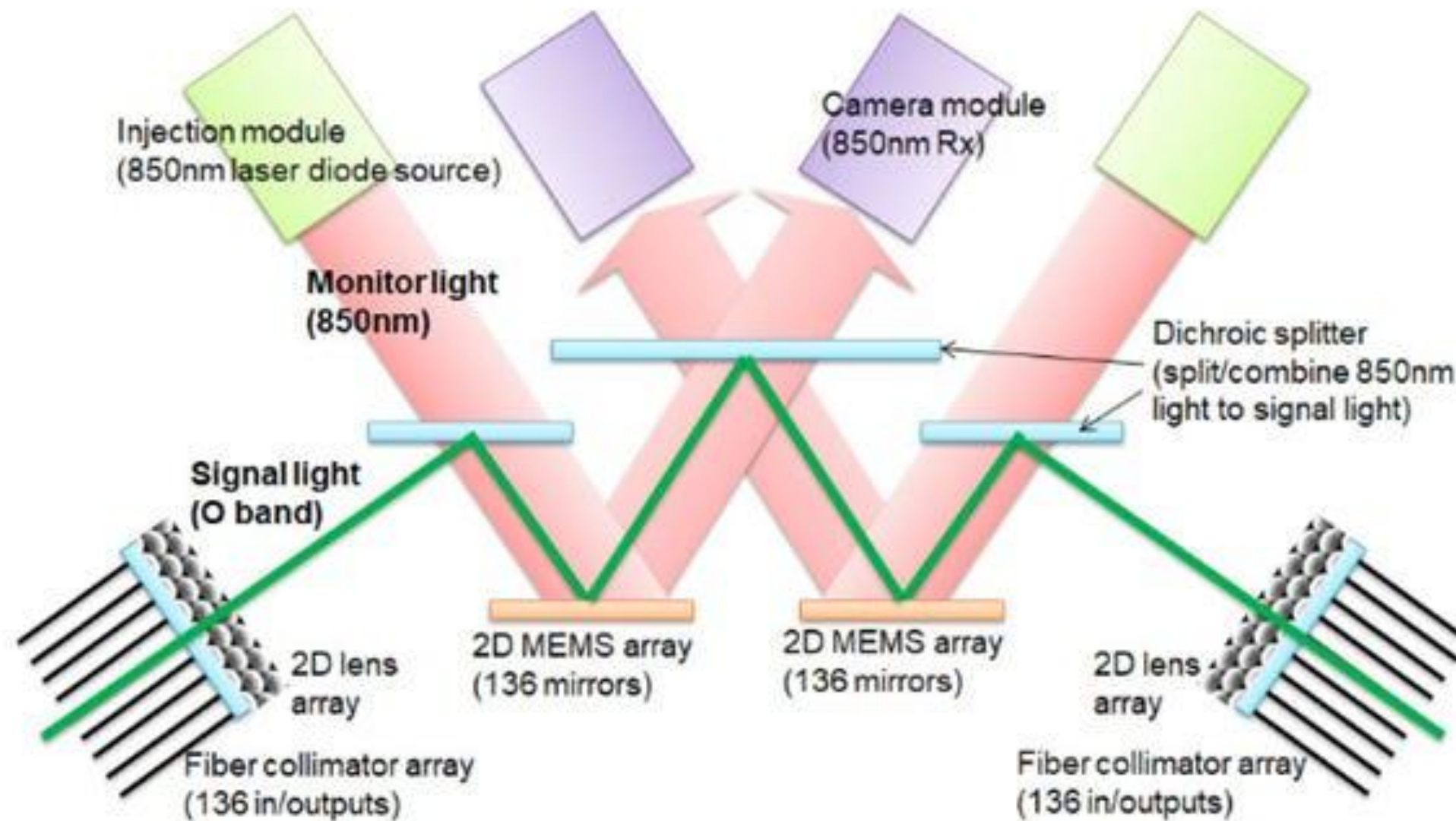


Nicole Morena, Cozy Kitchen, Blender.org

# Abstraction: Semiconductors


Semiconductors are an abstraction from the “physical” world to a “digital” one.

They aren't *necessarily* required.



# Abstraction Leak: Dynamic Power

Question: Do the contents of a tensor affects its performance?



```
zero_inputs = torch.zeros(N, N)
randn_inputs = torch.randn(N, N)
rand_inputs = torch.rand(N, N)
benchmark(zero_inputs)
benchmark(randn_inputs)
benchmark(rand_inputs)
```

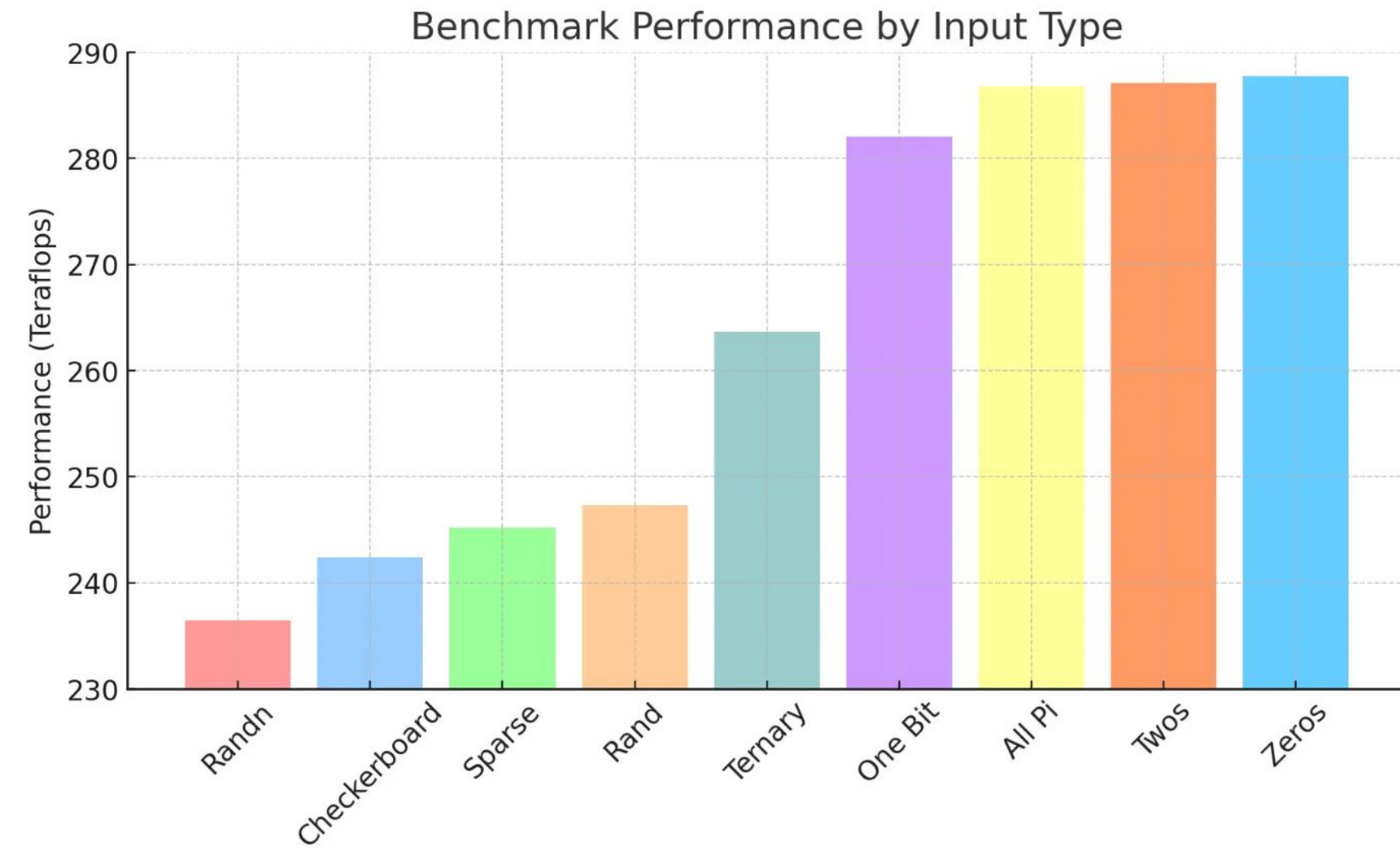
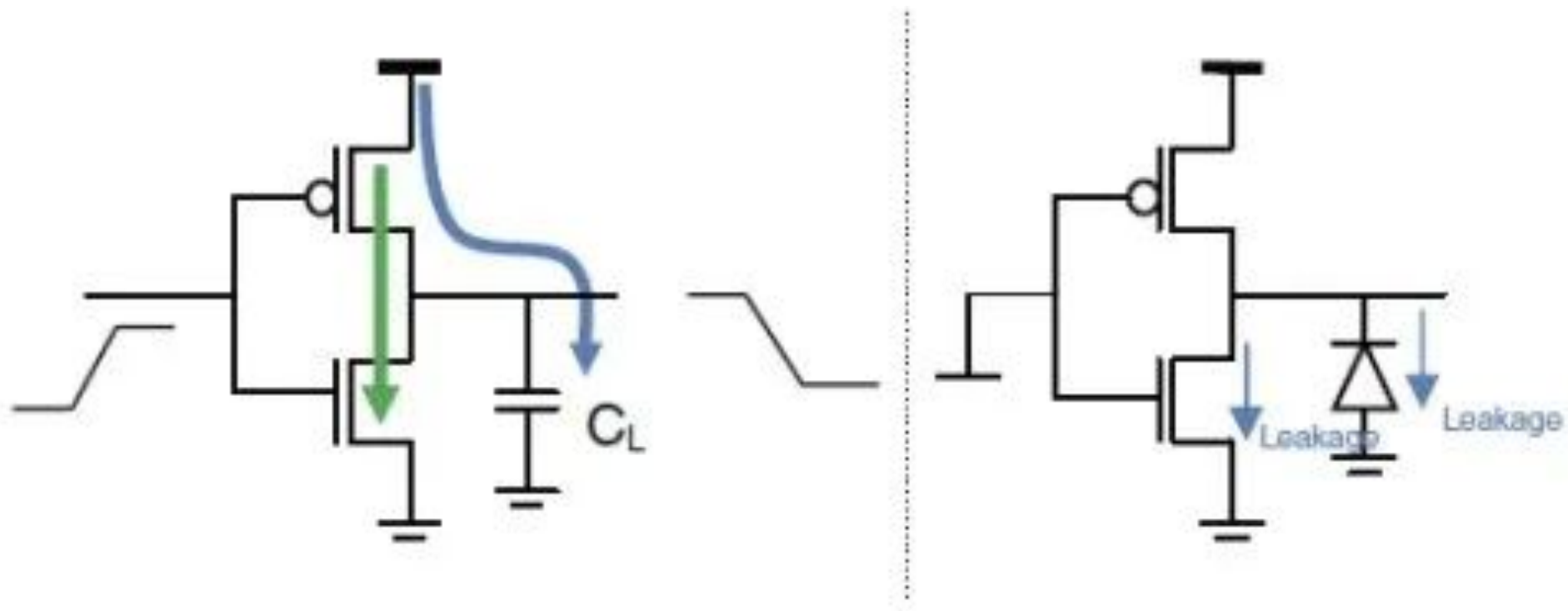
# Abstraction Leak: Dynamic Power

Question: Do the contents of a tensor affects its performance?

```
zero_inputs = torch.zeros(N, N)
randn_inputs = torch.randn(N, N)
rand_inputs = torch.rand(N, N)
benchmark(zero_inputs) # 295 TFL0PS
benchmark(randn_inputs) # 257 TFL0PS
benchmark(rand_inputs) # 268 TFL0PS
```

Answer: Yes they can!

# Abstraction Leak: Dynamic Power





# Abstraction: GPUs

A computer chip (like a GPU) is an abstraction over a bunch of transistors, turning it to a single “thing” from software’s perspective.

But... there are many alternative ways to arrange the transistors.

One interesting way that GPUs abstract over transistors is “floor sweeping”.

# Floor Sweeping

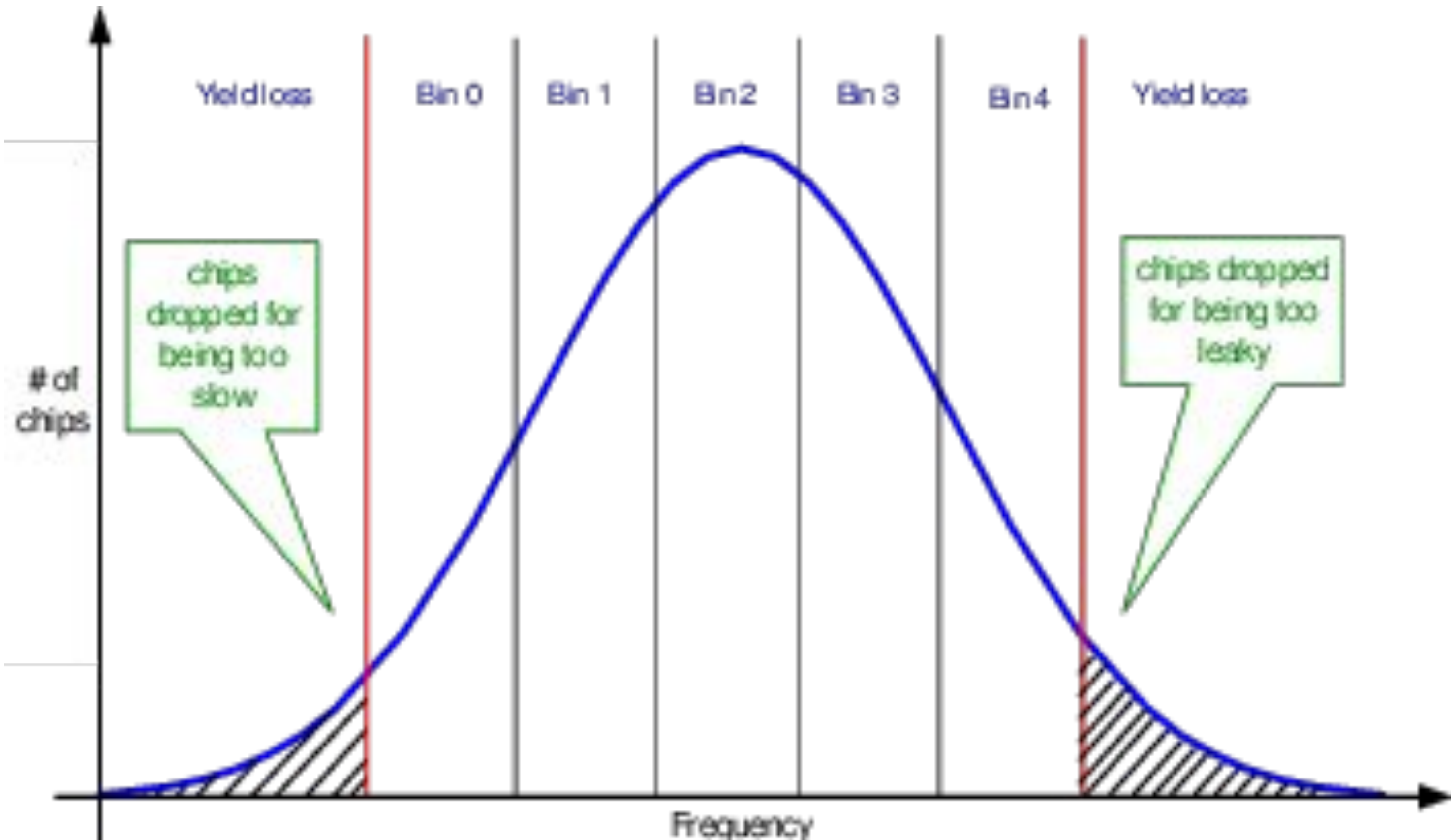
Basic problem:

You have 100 billion transistors in your GPU. Some of them did not come out “right”. What do you do with them?



Solution: “Disable” some portions that are too broken, choose some configuration that allows you to maximize performance and minimize “unusable” chips.

# Binning



# Abstraction Leak: Floor Sweeping

The **full implementation** of the GH100 GPU includes the following units:

- 8 GPCs, 72 TPCs (9 TPCs/GPC), 2 SMs/TPC, 144 SMs per full GPU
- 128 FP32 CUDA Cores per SM, 18432 FP32 CUDA Cores per full GPU
- 4 Fourth-Generation Tensor Cores per SM, 576 per full GPU
- 6 HBM3 or HBM2e stacks, 12 512-bit Memory Controllers
- 60 MB L2 Cache
- Fourth-Generation NVLink and PCIe Gen 5

The **NVIDIA H100 GPU with SXM5 board form-factor** includes the following units:

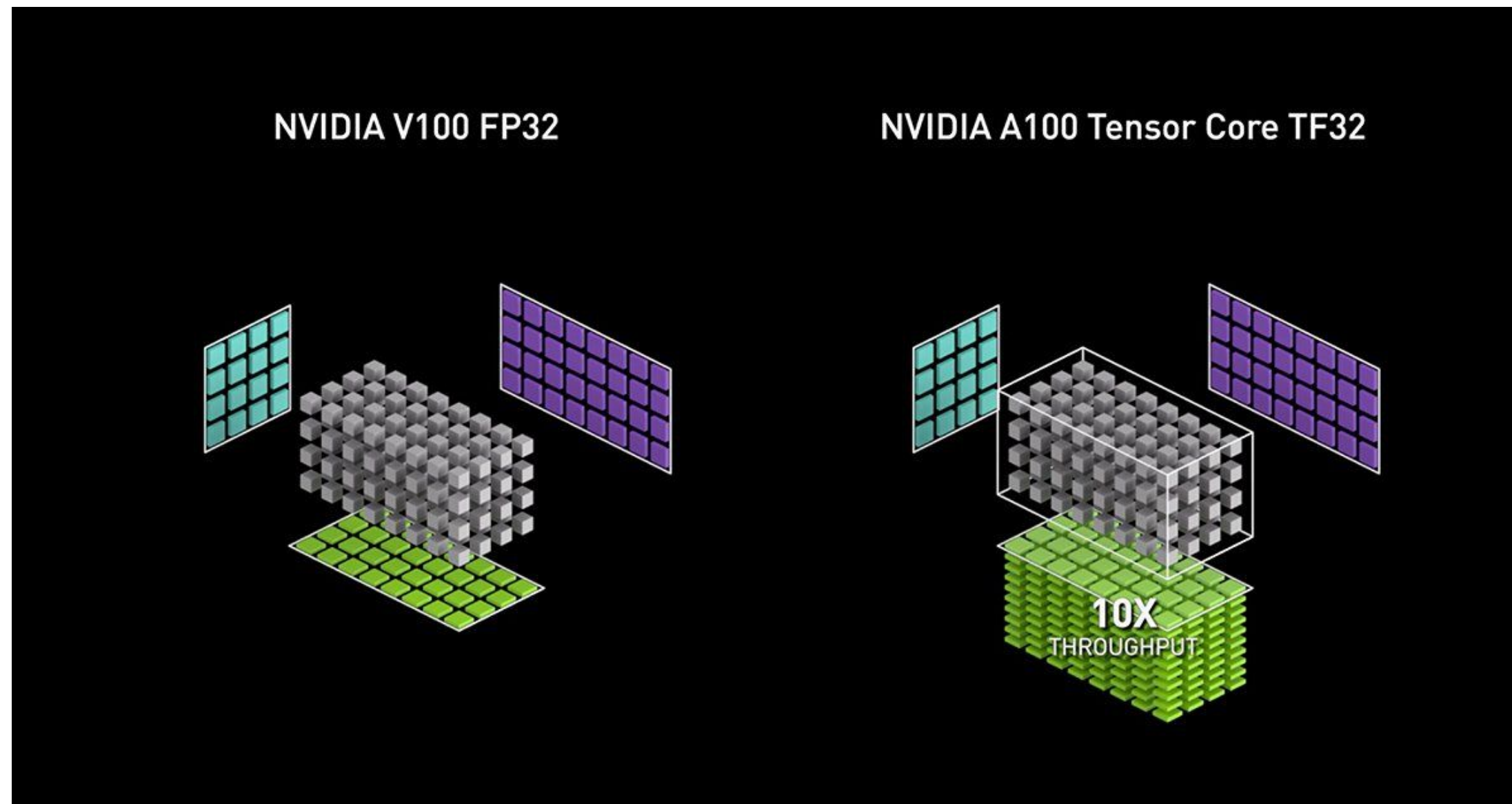
- 8 GPCs, 66 TPCs, 2 SMs/TPC, 132 SMs per GPU
- 128 FP32 CUDA Cores per SM, 16896 FP32 CUDA Cores per GPU
- 4 Fourth-generation Tensor Cores per SM, 528 per GPU
- 80 GB HBM3, 5 HBM3 stacks, 10 512-bit Memory Controllers
- 50 MB L2 Cache
- Fourth-Generation NVLink and PCIe Gen 5

The **NVIDIA H100 GPU with a PCIe Gen 5 board form-factor** includes the following units:

- 7 or 8 GPCs, 57 TPCs, 2 SMs/TPC, 114 SMs per GPU
- 128 FP32 CUDA Cores/SM, 14592 FP32 CUDA Cores per GPU
- 4 Fourth-generation Tensor Cores per SM, 456 per GPU
- 80 GB HBM2e, 5 HBM2e stacks, 10 512-bit Memory Controllers
- 50 MB L2 Cache
- Fourth-Generation NVLink and PCIe Gen 5

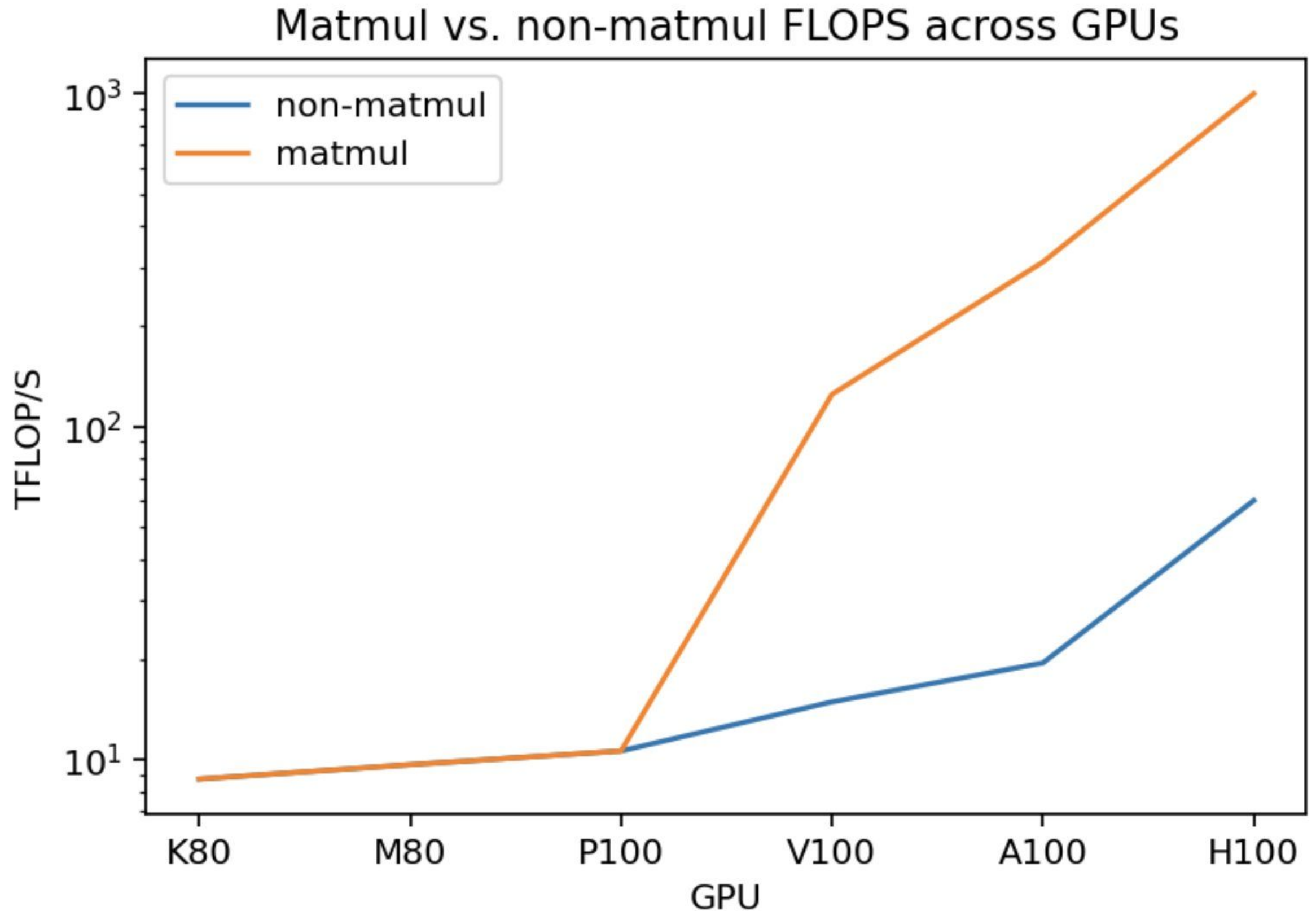
# Abstraction Backpressure: Tensor Cores

- “Hardware built for machine learning” i.e. “hardware built to do matrix multiplication”.
- More than just inherent matmul properties, there’s literally a hardware op for them.



# Tensor-Core Performance

- Introduced tensor cores
- Drastically increased ML performance of Nvidia GPUs.



# Abstraction Constraint: Hardware Lottery

- No matter how clever you are, anything that can't be fit into a matmul will be 15x slower than a matmul.
- This is an example of a so-called “hardware lottery”.
- Many other operations that look similar that'll never be similarly accelerated.
- Nearest neighbor with L2 similarity  $\gg$  L1 similarity



**Sasha Rush**

@srush\_nlp



Btw, one reason this is underused is because log-space is still hard and slow in 2021.

If someone (@NVIDIAAI?), would implement a fast log-space MM [(A[..., None] + B[None]).logsumexp(-2)] it would be amazing. Been failing to do this for years, Cutlass is hard 😞

# You don't need to worry about every abstraction!

- If you're working far enough from the hardware, you don't need to worry about whether tensor-cores are the "right" abstraction - you can't do anything about it anyways!
- Perhaps if you're at a company building their own chips it may be worth talking to the hardware folk about having support for operations you want (i.e. mx4, fp8, etc.)



Prompting/End User Applications

Machine Learning Models

ML Software

ML Hardware

Semiconductors

Physics

# We have a GPU with thousands of cores. What now?

Everybody knows how to write code for a single processor - everything you write typically just runs on one processor.

But what about when we have thousands of cores? Writing parallel code is so much harder... what can we even do?



# Abstraction: Array Programming

Goal: Start with operations on arrays - abstract away all details of how your “problem” gets mapped to the underlying hardware.

This works well regardless of the underlying hardware!

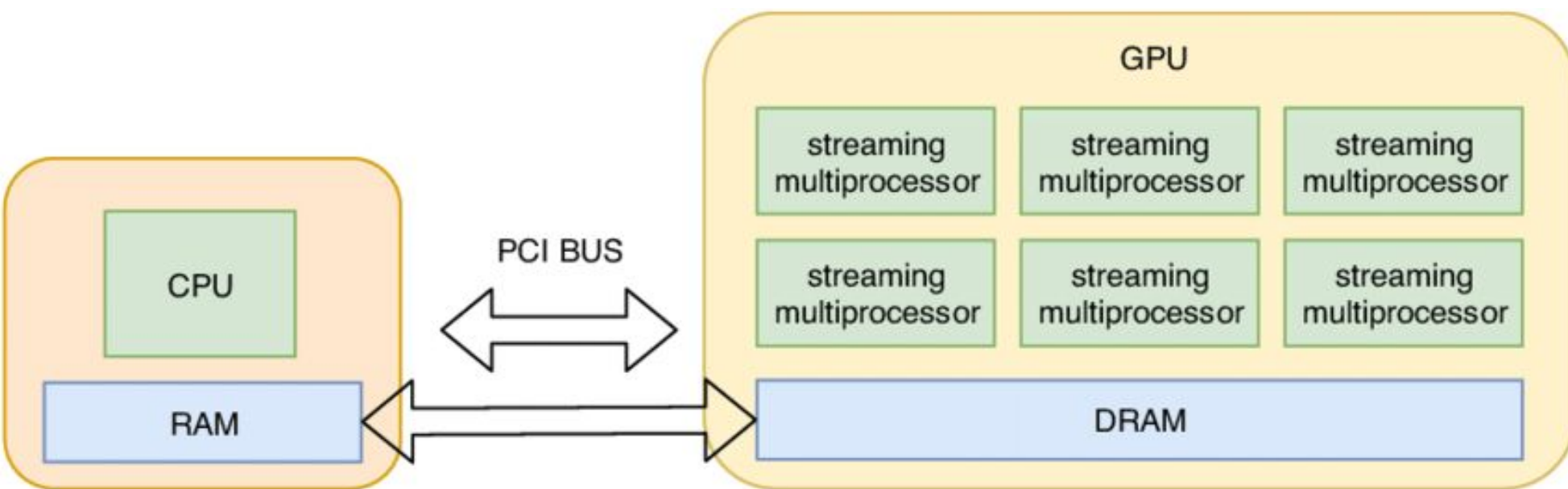


```
torch.add(A, B) # Parallelizes to hardware  
torch.mm(A, B) # Maps to some efficient matmul kernel
```

# Abstraction: Interpreter-like Execution

Goal: Abstract away GPU execution details while being performant

When ML models were completely dominated by matmuls, “dynamic” graphs were a much nicer abstraction for users to deal with that was just as performant.



# Abstraction: Interpreter-like Execution

Very “simple” abstraction.

Not only easier to debug, also much more “composable”.

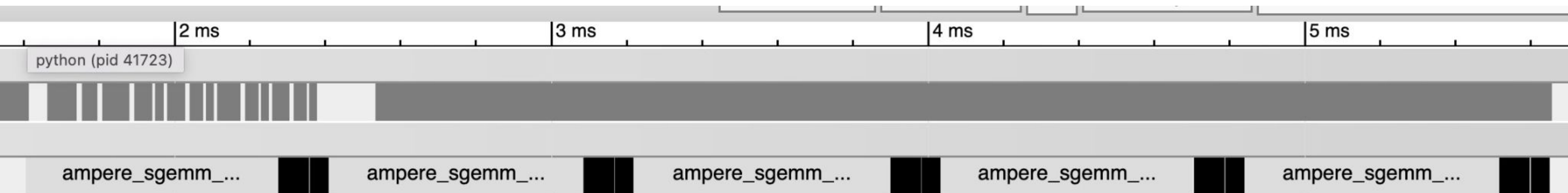


```
a = ...  
torch.mm(a, a) # This immediately runs on the GPU  
a + a          # This immediately runs on the GPU
```

# Abstraction Leak: Non-Matmul Performance

**Assumption: Neural Networks spend the vast majority of their time doing matmuls**

We're only spending 15% of our time doing anything that's not a matmul.



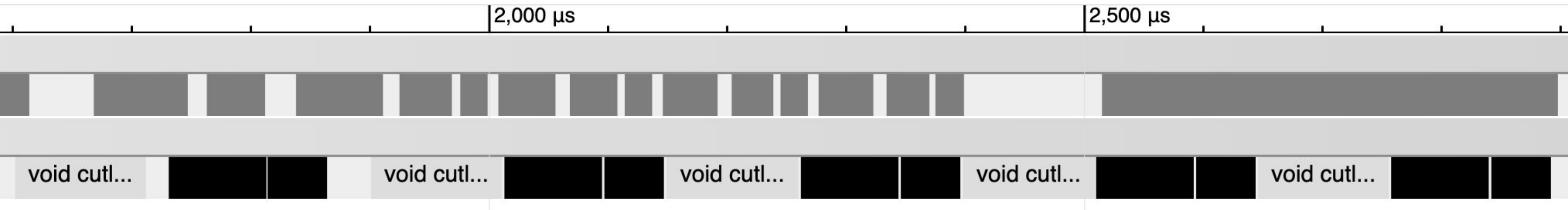
# Hardware constraints changed!

~~Assumption: Neural Networks spend the vast majority of their time doing matmuls~~

All of a sudden, matmuls get 10x faster, while everything else stays at similar speed.

We're now spending 60% of our time doing non-matmul things!

So there's a lot more room to do optimizations.



# Dynamic Programming Model with Static Capture

`tf.function`, `jax.jit`, `torch.compile`, etc.

The “programming model” is that it’s interpreter-like execution.

But... we “capture” your function in some manner to turn it into a static graph.

There’s a variety of ways of doing this, but the 2 main approaches folks use are:

1. Tracing (`jax.jit`, `torch.fx`)
2. Bytecode (`torch.compile`)





# Abstraction “Leak”: Peak Performance

Another important change over the last 5 years has been the consolidation of architectures and the focus on scale.

Whereas previously, 90% of the performance was usually fine, folks often want 100% of the performance nowadays.

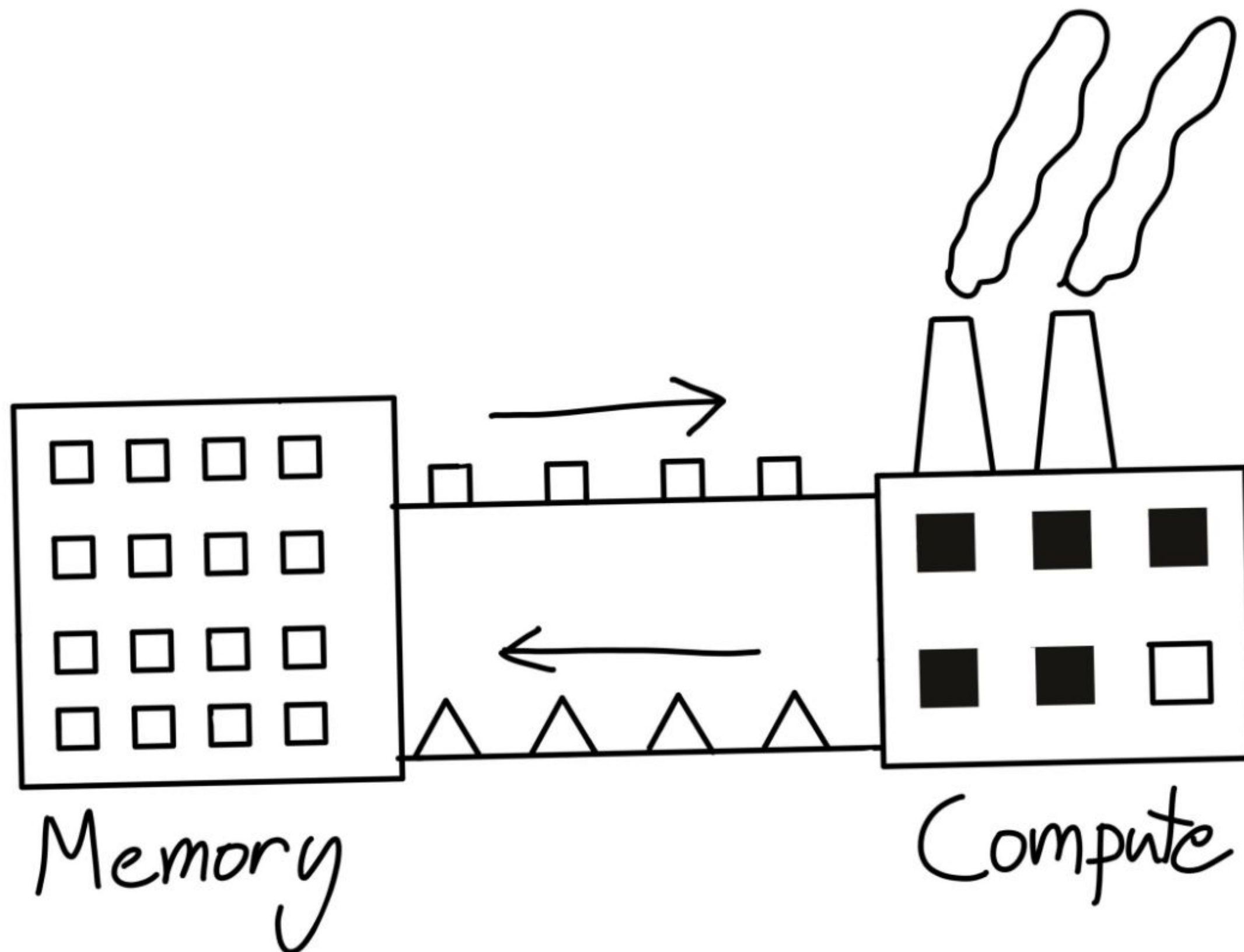
So... a lot more users “care” about performance nowadays.

But how do you start thinking about Deep Learning performance?

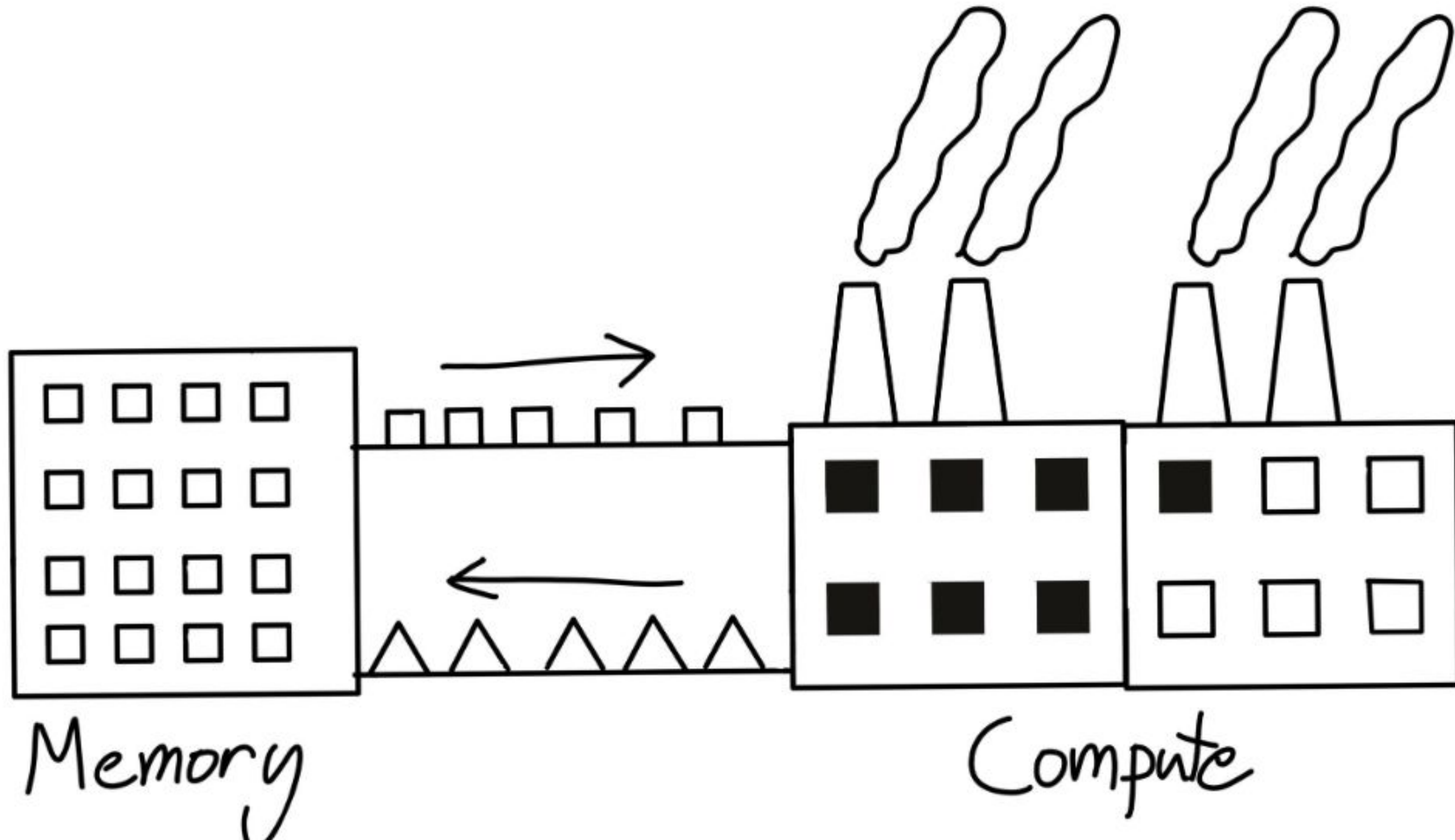
# Primer on Deep Learning Performance: Where are you spending your time?

1. Compute: Time spent on your GPU doing actual computation
2. Memory: Time spent transferring tensors within a GPU.
3. Overhead: Everything else

# Compute



# Compute is often not the limiting factor



*Table 1. Proportions for operator classes in PyTorch.*

---

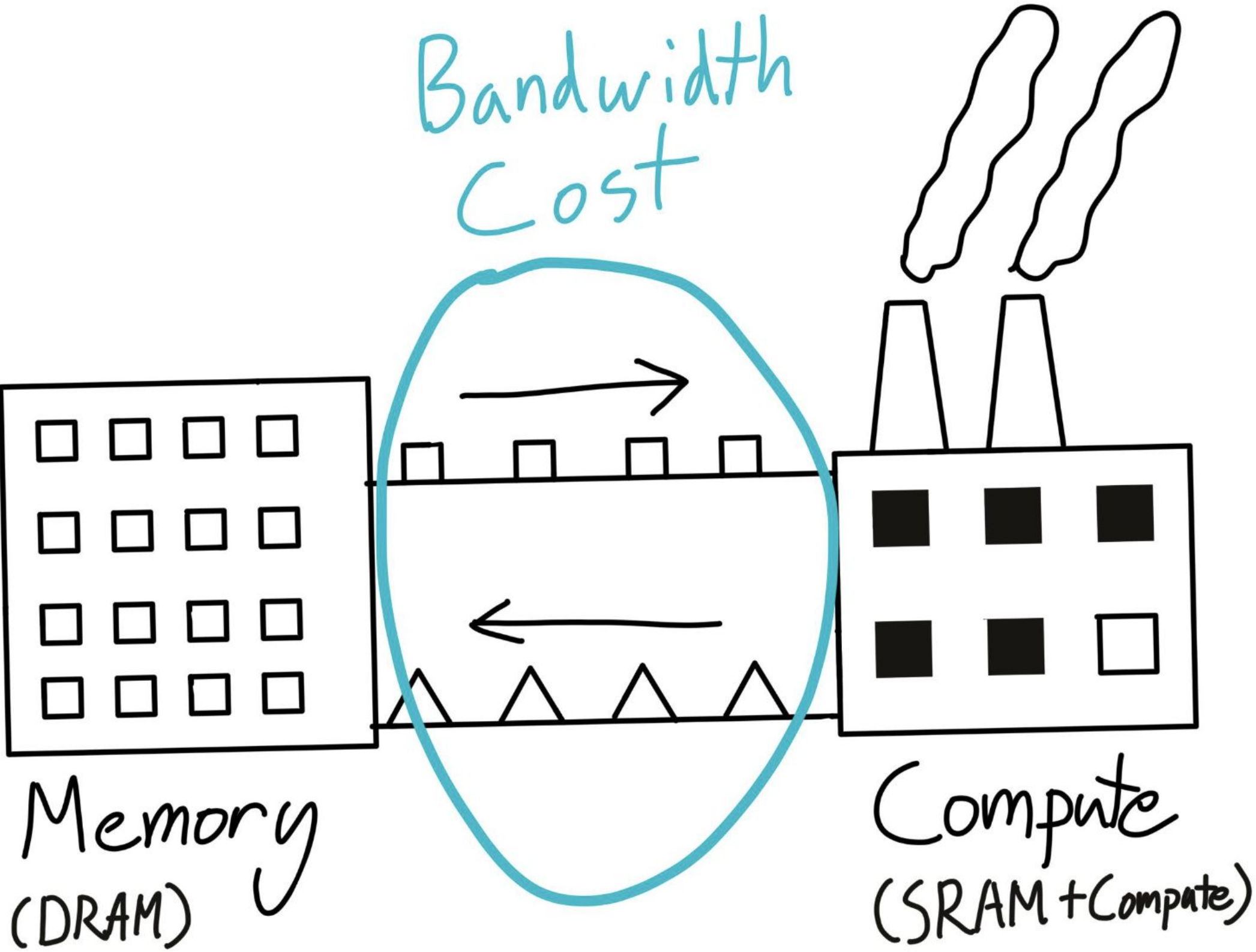
Operator class	% flop	% Runtime
$\Delta$ Tensor contraction	99.80	61.0
$\square$ Stat. normalization	0.17	25.5
$\circ$ Element-wise	0.03	13.5

---

# Time on Compute == Time doing Matmuls

<b>H100 SXM</b>	
<b>FP64</b>	34 teraFLOPS
<b>FP64 Tensor Core</b>	67 teraFLOPS
<b>FP32</b>	67 teraFLOPS
<b>TF32 Tensor Core*</b>	989 teraFLOPS
<b>BFLOAT16 Tensor Core*</b>	1,979 teraFLOPS
<b>FP16 Tensor Core*</b>	1,979 teraFLOPS
<b>FP8 Tensor Core*</b>	3,958 teraFLOPS
<b>INT8 Tensor Core*</b>	3,958 TOPS

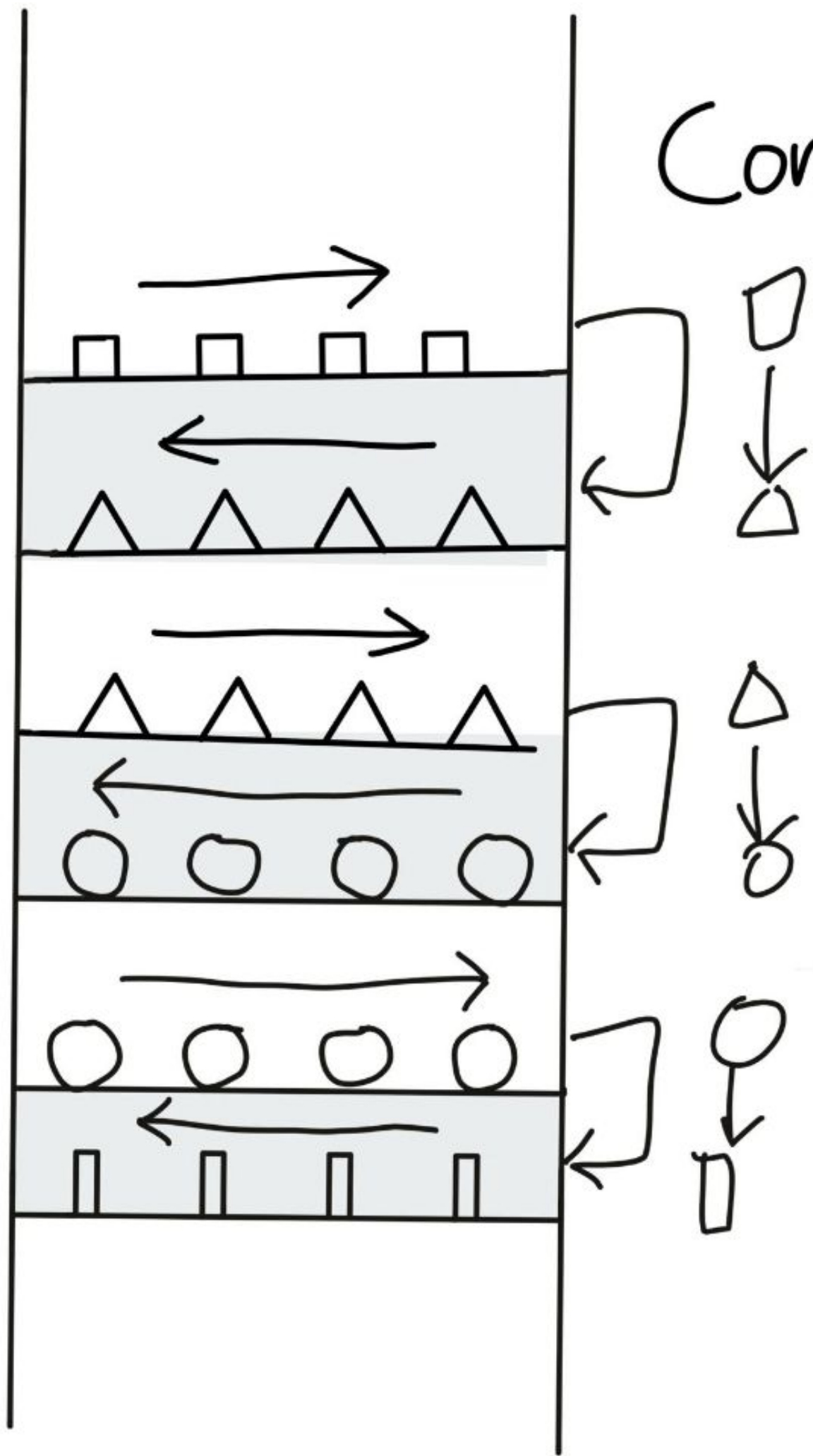
# Memory





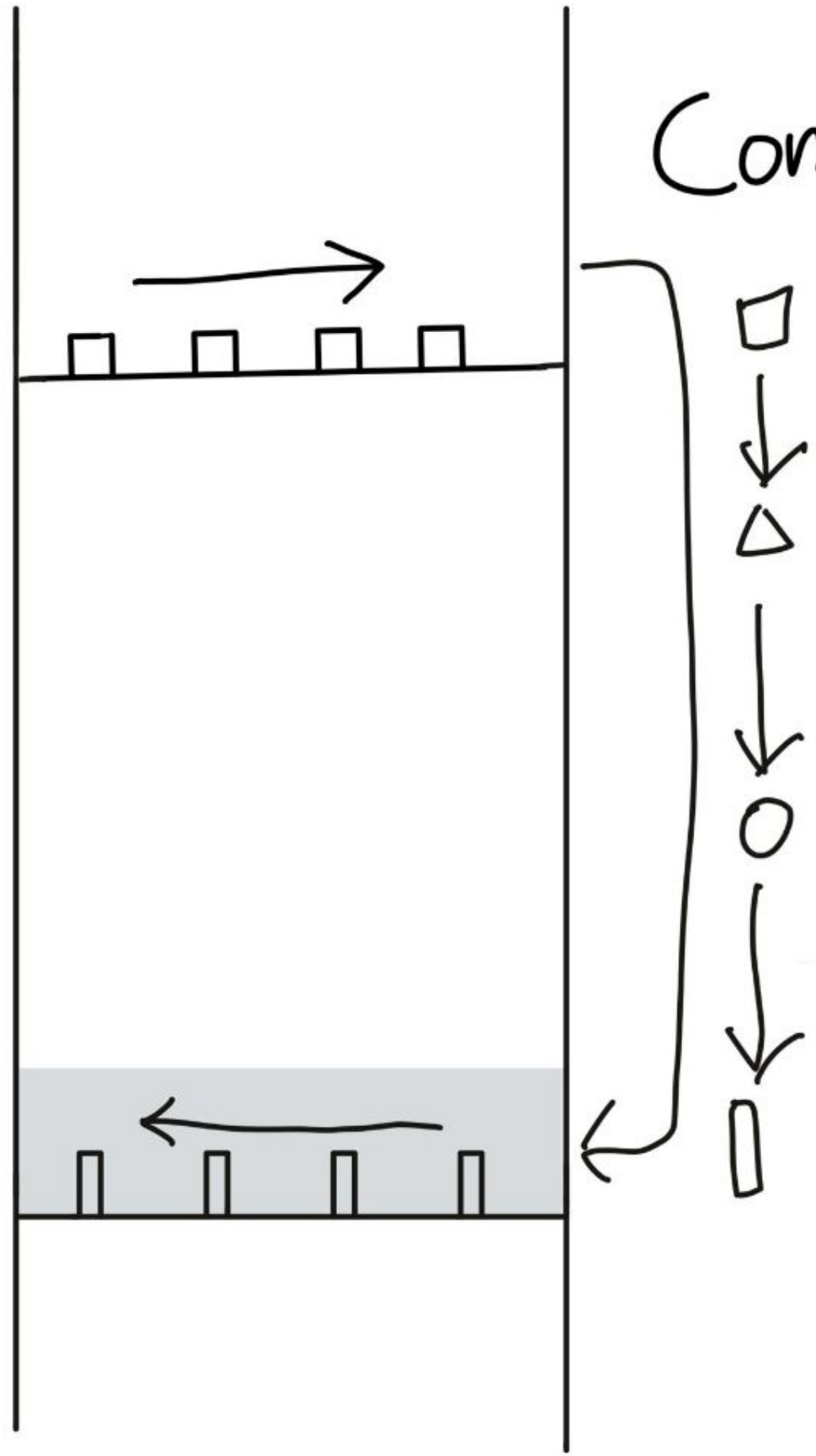
Memory

Compute



Memory

Compute



# How many multiplies do you need to fuse together before the kernel takes twice as long?

A100:

10 Teraflops of multiply compute

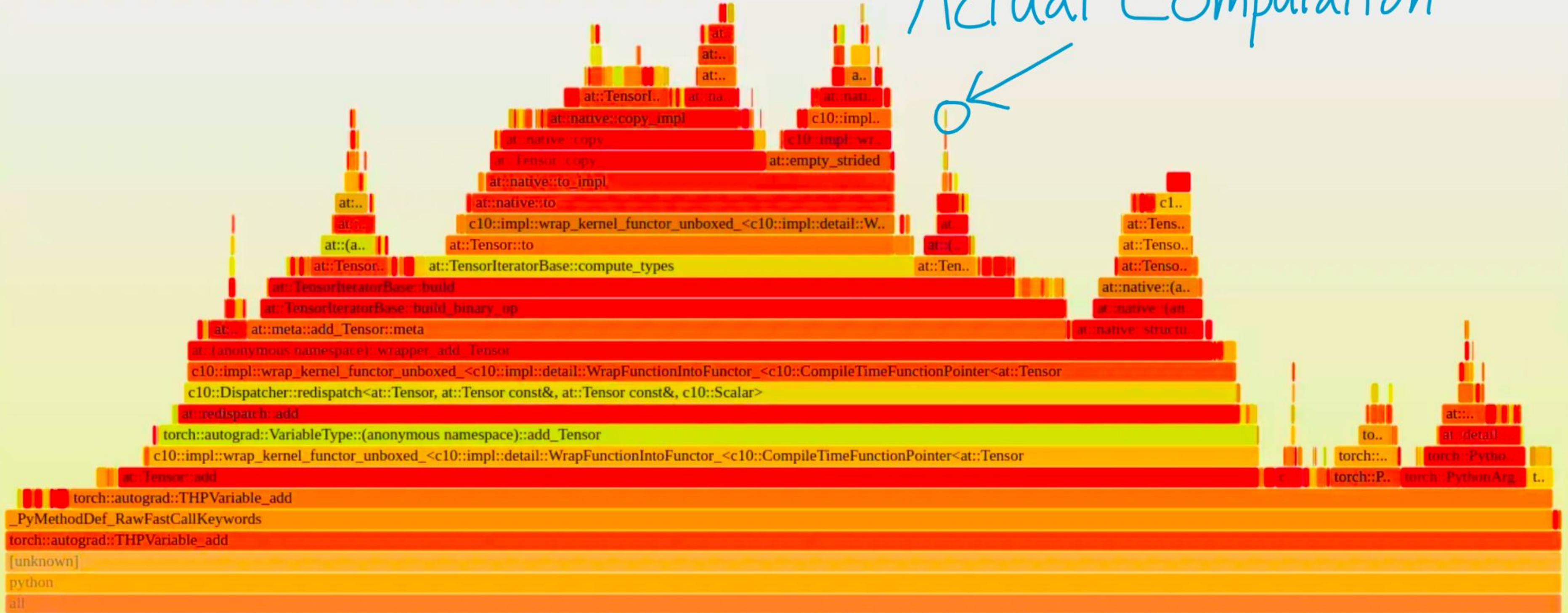
2 Terabytes of memory bandwidth

float32 computation

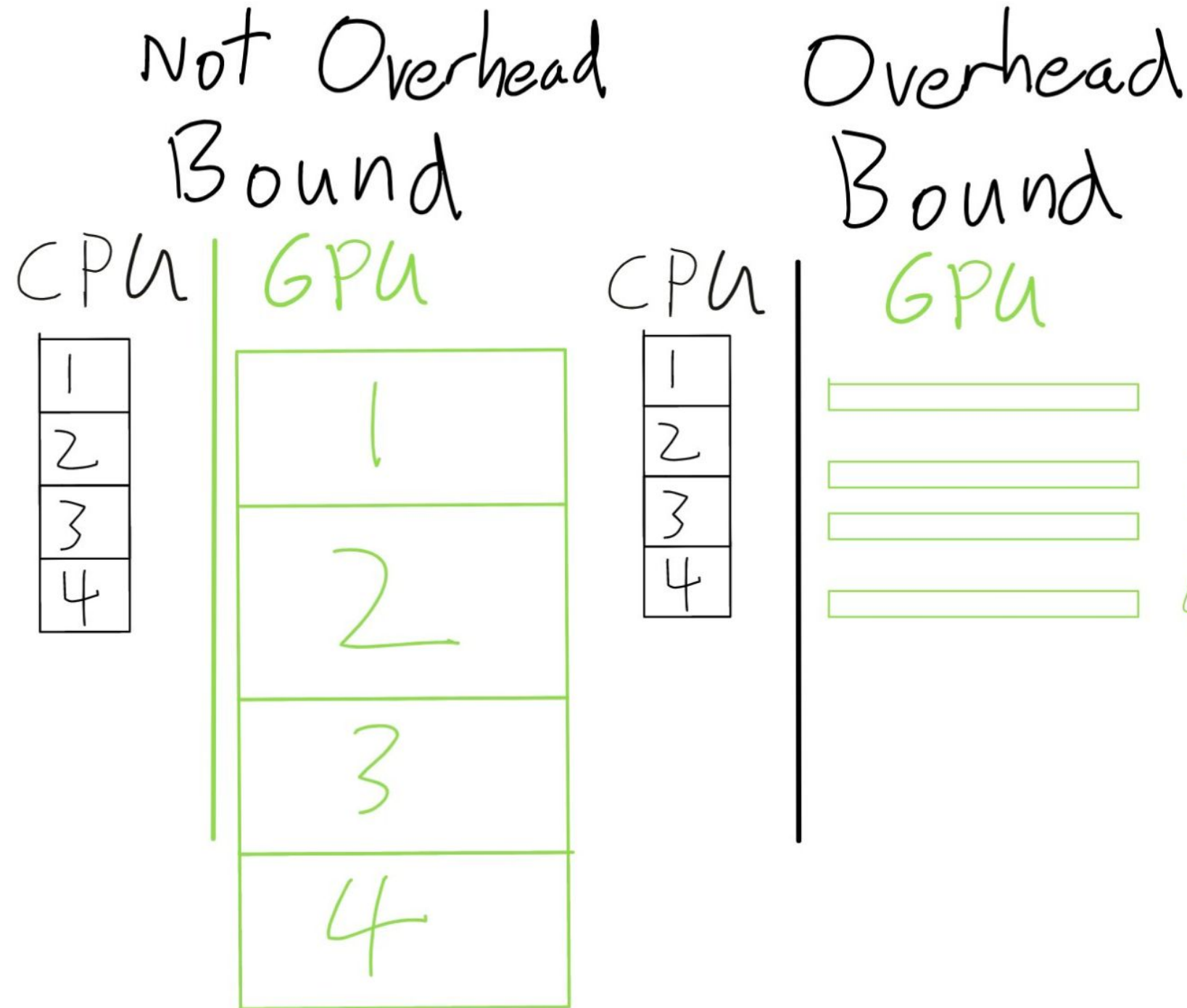
```
def f(x: Tensor[N]):  
    for _ in range(repeat):  
        x = x * 2  
    return x
```

<https://claude.site/artifacts/849c93d3-115b-4e69-9e00-9d5bcc3f8389>

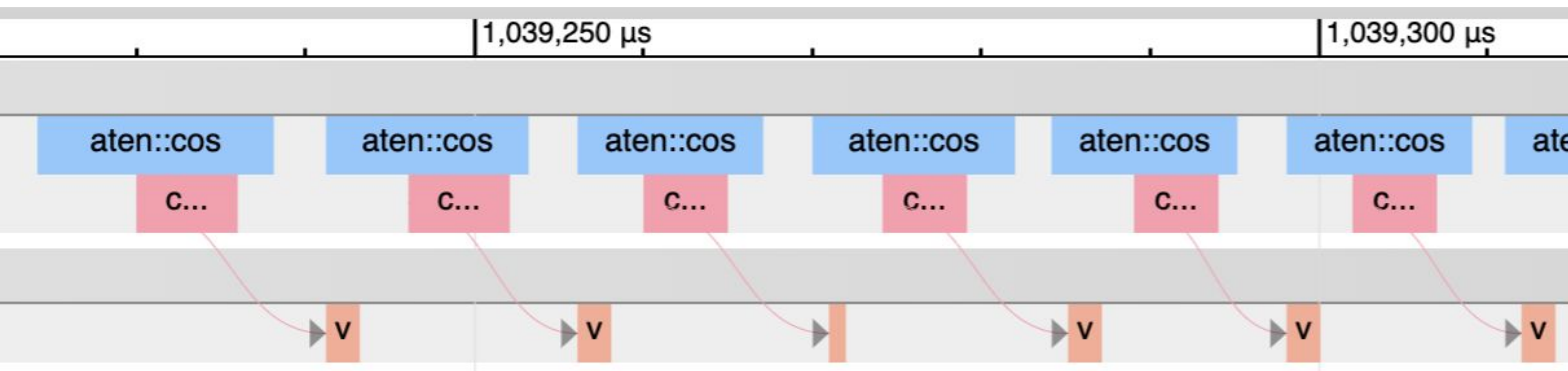
Actual Computation

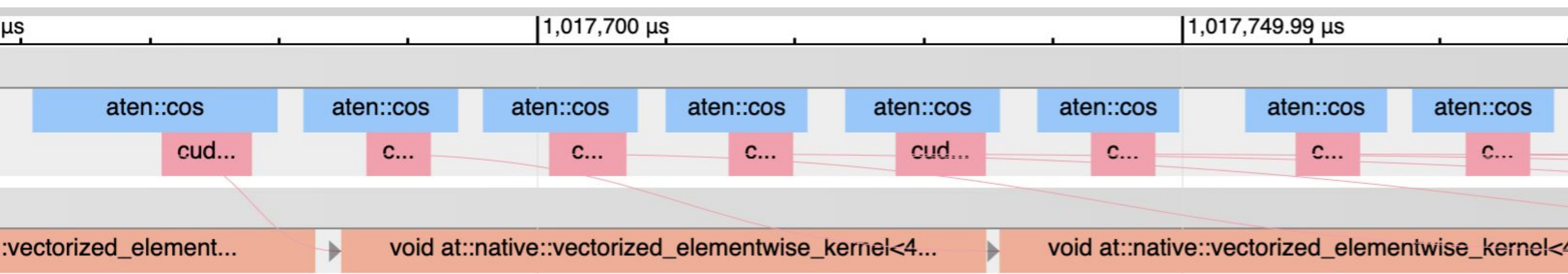


# Parallelism among CPU and GPU



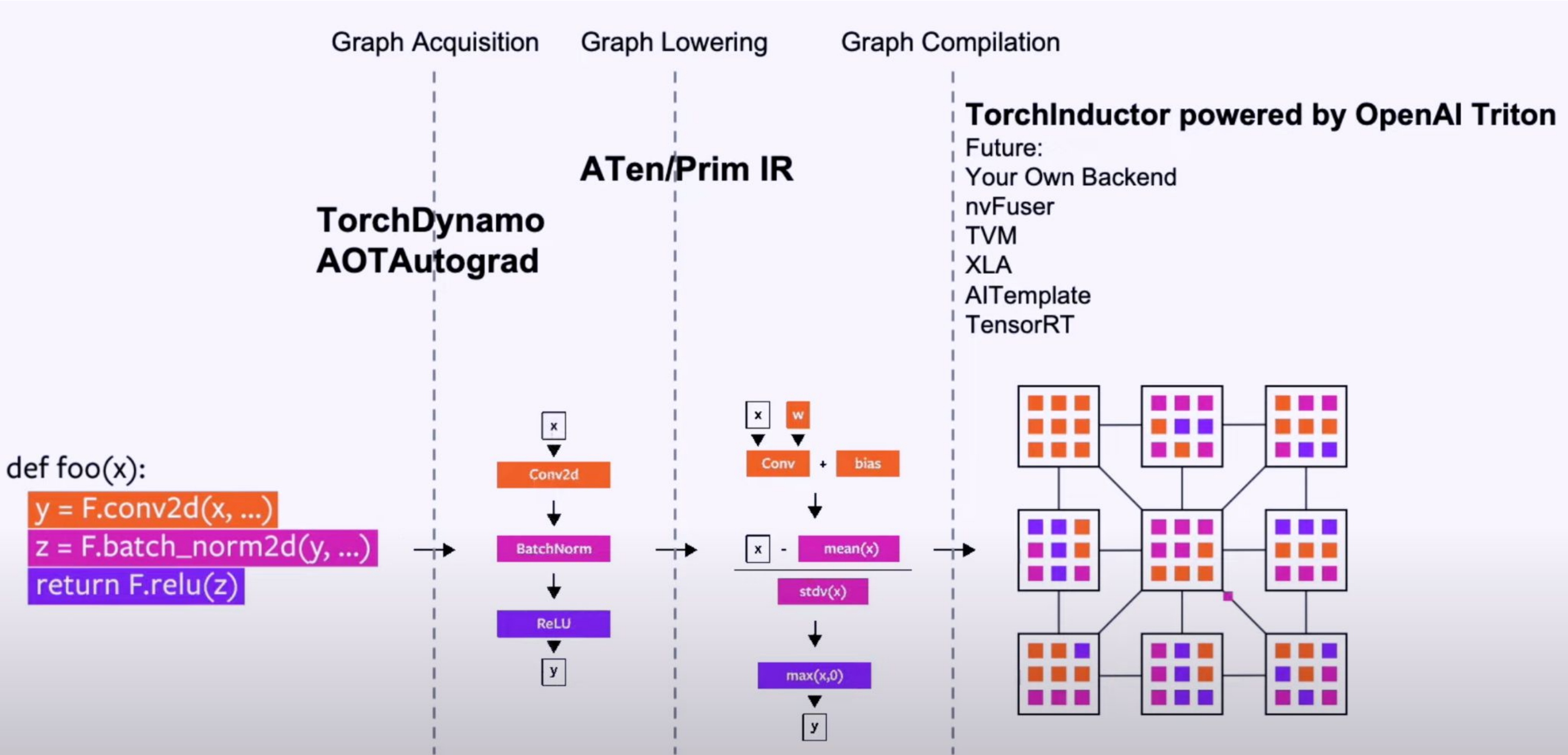
*If our GPU operators are big enough, then our CPU can run ahead of the GPU (and thus the CPU overhead is irrelevant). On the other hand, if our GPU operators are too small, then our GPU is going to spend most of its time as an expensive paperweight.*





# Can we abstract over these details?

To some extent... yes! This is what compilers like TorchInductor or XLA do.





# What about FlashAttention?

Why didn't ML frameworks just "automatically" do FlashAttention for us?

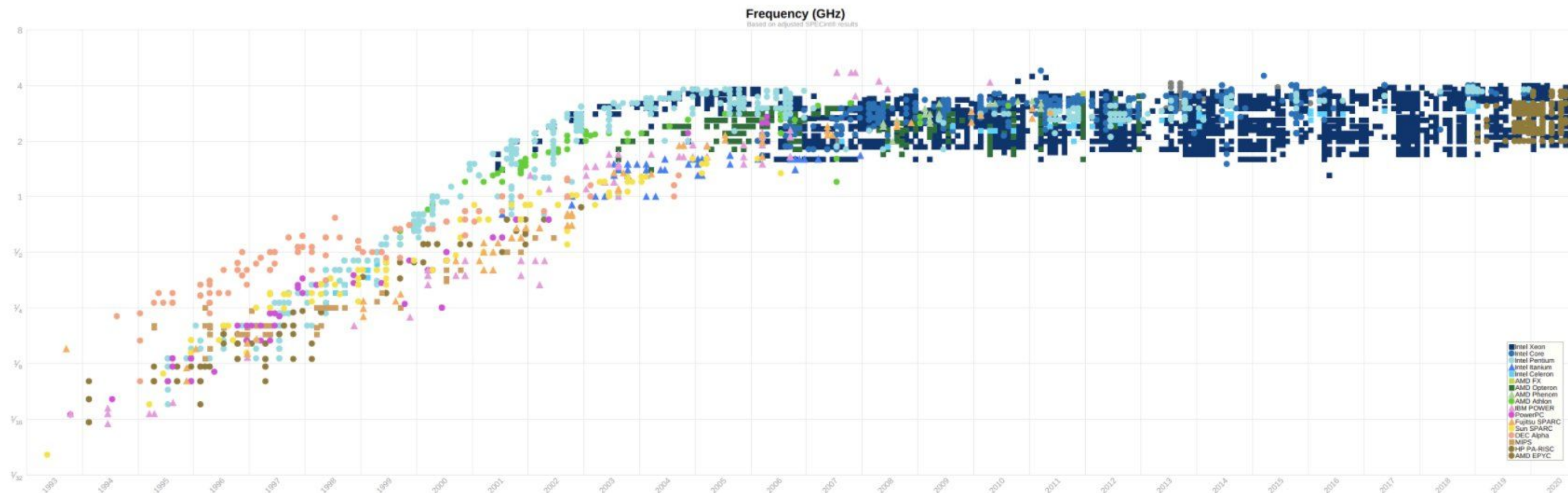
Can compilers just handle *all* optimization for us?

To answer this, let's look at an (old-ish) analogue - autovectorization/autoparallelism.

# Autoparallelism/autovectorization

People saw the death of Dennard scaling coming. But everybody was writing single-threaded code. How would we possibly take advantage of parallelism?

Enter... Autoparallelism, where they mainly focused on taking Fortran code and making it automatically parallel.



Transistors got exponentially faster until the moment they didn't.

# “Autovectorization is not a programming model”

The problem with an auto-vectorizer is that as long as vectorization can fail (and it will), then if you're a programmer who actually cares about what code the compiler generates for your program, you must come to deeply understand the auto-vectorizer. Then, when it fails to vectorize code you want to be vectorized, you can either poke it in the right ways or change your program in the right ways so that it works for you again. This is a horrible way to program; it's all alchemy and guesswork and you need to become deeply specialized about the nuances of a single compiler's implementation—something you wouldn't otherwise need to care about one bit.

And God help you when they release a new version of the compiler with changes to the auto-vectorizer's implementation.

Autovectorization isn't even an abstraction!

# Implicit Parallelism - SIMT (CUDA), MPI, etc.

Program from the perspective  
of one thread.

Parallelism is \*implicit\* in the programming  
model

Very difficult to screw it up.

Autoparallelization faded away in the  
90s and 00s

```
BLOCK = 512

# This is a GPU kernel in Numba.
# Different instances of this
# function may run in parallel.
@jit
def add(X, Y, Z, N):
    # In Numba/CUDA, each kernel
    # instance itself uses an SIMT execution
    # model, where instructions are executed in
    # parallel for different values of threadIdx
    tid = threadIdx.x
    bid = blockIdx.x
    # scalar index
    idx = bid * BLOCK + tid
    if id < N:
        # There is no pointer in Numba.
        # Z,X,Y are dense tensors
        Z[idx] = X[idx] + Y[idx]

...
grid = (ceil_div(N, BLOCK),)
block = (BLOCK,)
add[grid, block](x, y, z, x.shape[0])
```

# It's quite difficult to generate FlashAttention from scratch

1. Back-to-back matmul
2. Online softmax requires a mathematical rewrite
3. Backwards also requires a mathematical rewrite
4. Typically requires some amount of structured sparsity

That's why we have...

`F.scaled_dot_product_attention!!!`

But... people keep coming out with new stuff...

PrefixLM

Transfusion

Sliding Window  
Attention

Alibi

TreeAttention

Softcapping

Relative Positional  
Encoding

Causal

PagedAttention

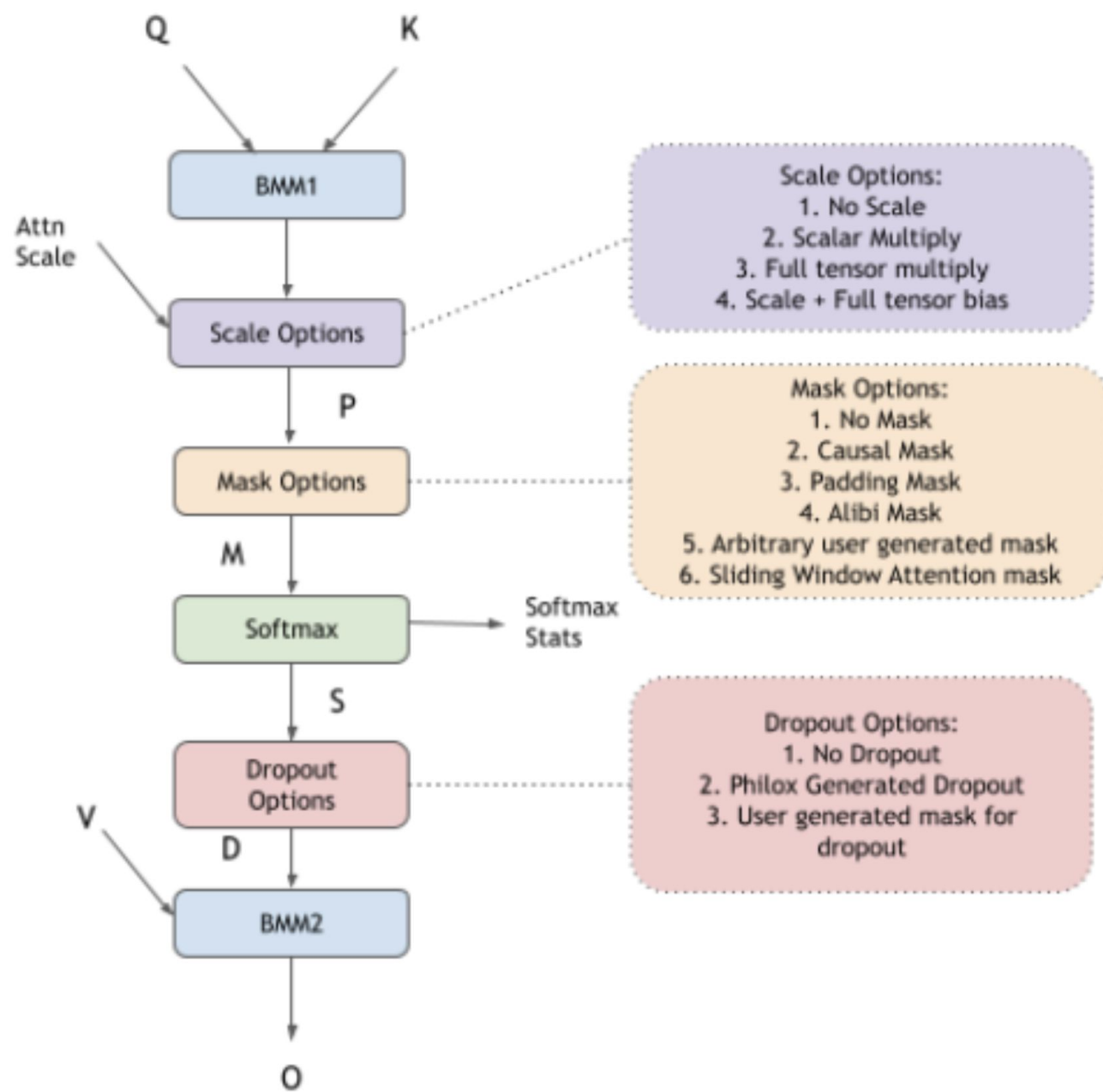
Jagged Sequences

Neighborhood  
Attention

GraphAttention

# Attention operators accumulate kwargs at a worrying pace

```
def _flash_attn_forward(  
    q, k, v, dropout_p, softmax_scale, causal, window_size, softcap, alibi_slopes, return_softmax  
):
```



```
def flash_attn_with_kvcache(  
    q,  
    k_cache,  
    v_cache,  
    k=None,  
    v=None,  
    rotary_cos=None,  
    rotary_sin=None,  
    cache_seqlens: Optional[Union[(int, torch.Tensor)]] = None,  
    cache_batch_idx: Optional[torch.Tensor] = None,  
    block_table: Optional[torch.Tensor] = None,  
    softmax_scale=None,  
    causal=False,  
    window_size=(-1, -1), # -1 means infinite context window  
    rotary_interleaved=True,  
    alibi_slopes=None,  
):
```

# Even with all the added kwargs, it's not enough...



**Ben (e/treats)**

@andersonbcdefg

Subscribe



is there still not flash attn for prefixlm? someone who knows CUDA / triton should have banged this out by now right where can i find it

8:07 AM · Jun 12, 2024 · **32K** Views



9



3



33



14



The other striking thing is how little support these codebases have for large scale encoder-decoder training or even prefixLM training. To that end, even flash attention has consistently declined to provide support for prefixLM training (i.e., custom masks) despite reasonable demand on their github issues for whatever reason.



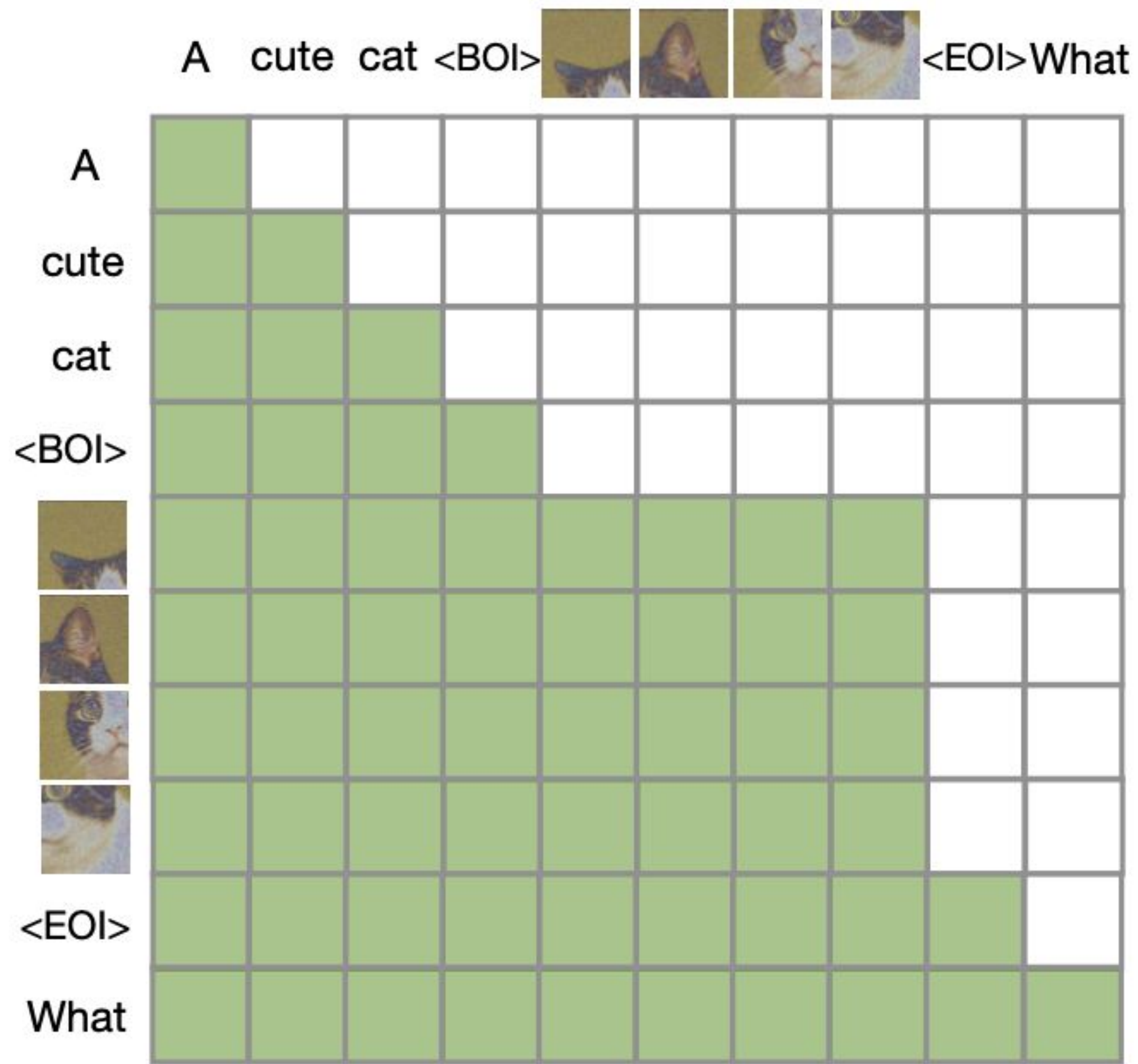
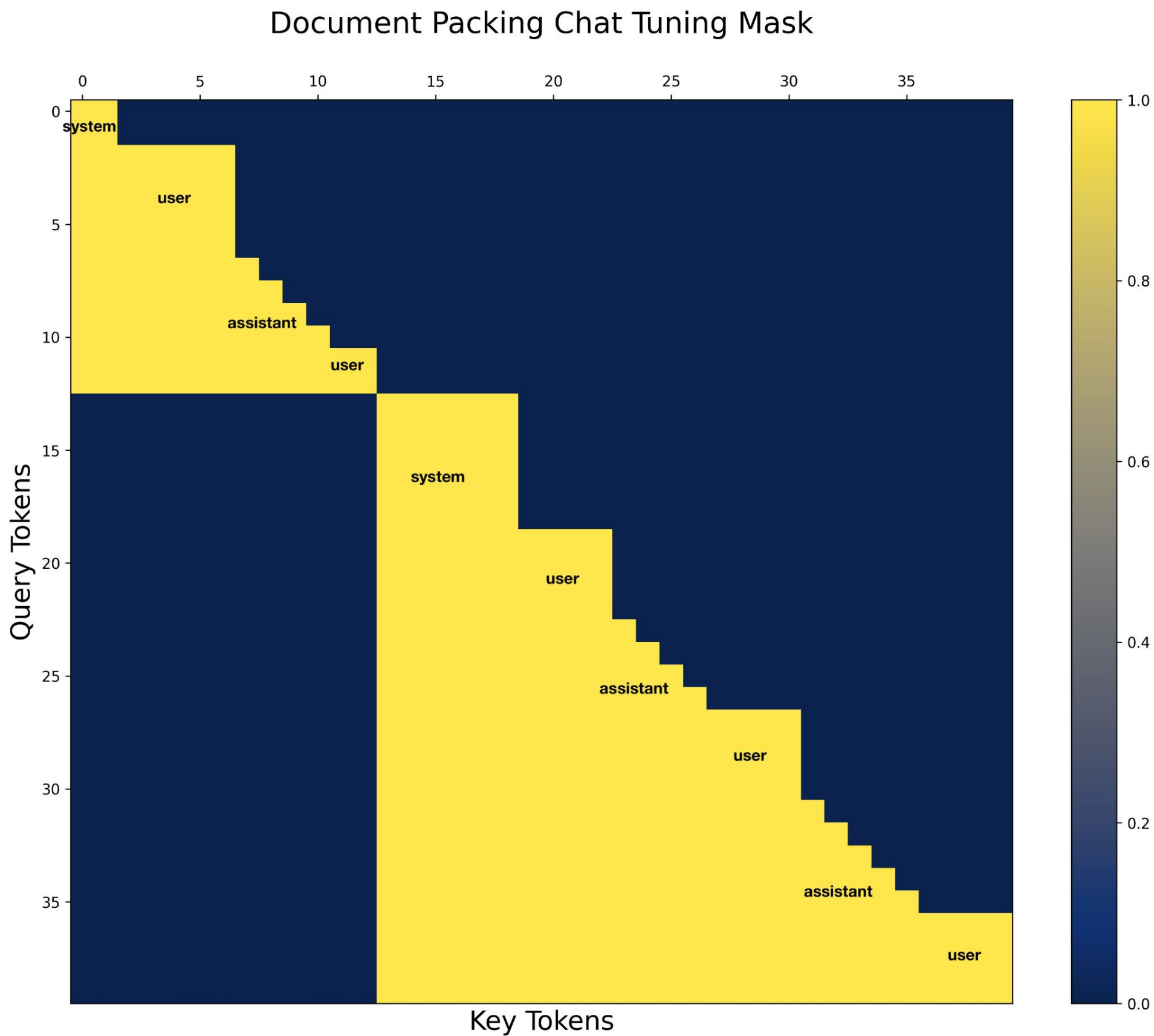


Figure 4: Expanding on the causal mask, Transfusion allows patches of the same image to condition on each other.

# So, compilers can't generate it, and it's painful to do modifications by hand. Are we screwed?

**Custom Kernels that are impossible for torch.compile to generate from scratch**

Decomposes into

**Handwritten/complicated  
FlashAttention kernel**

**+**

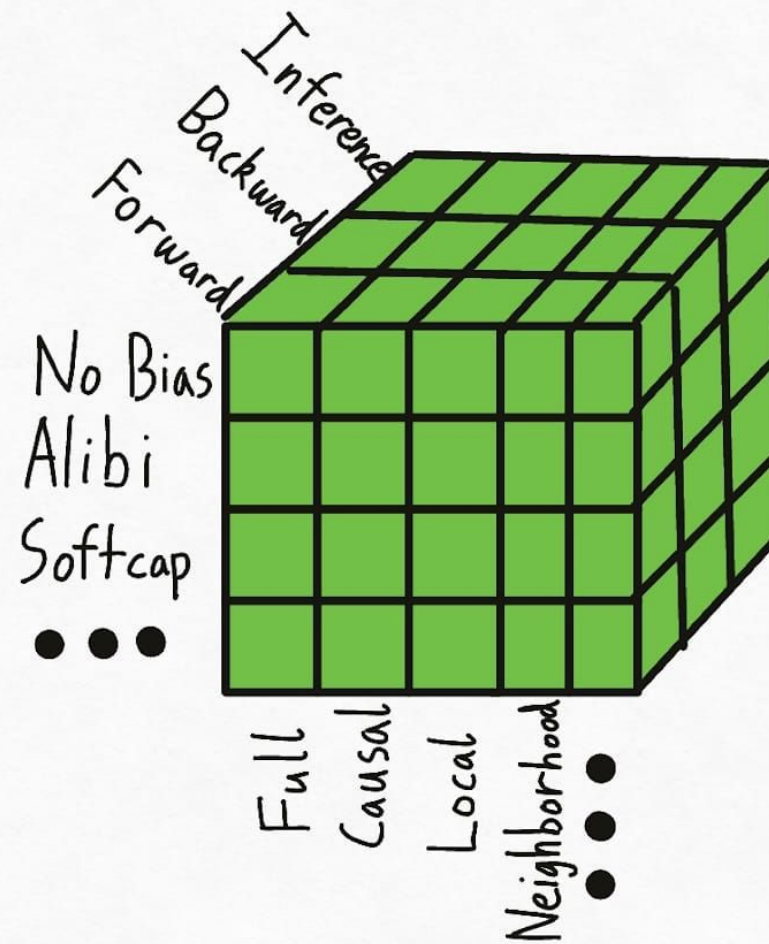
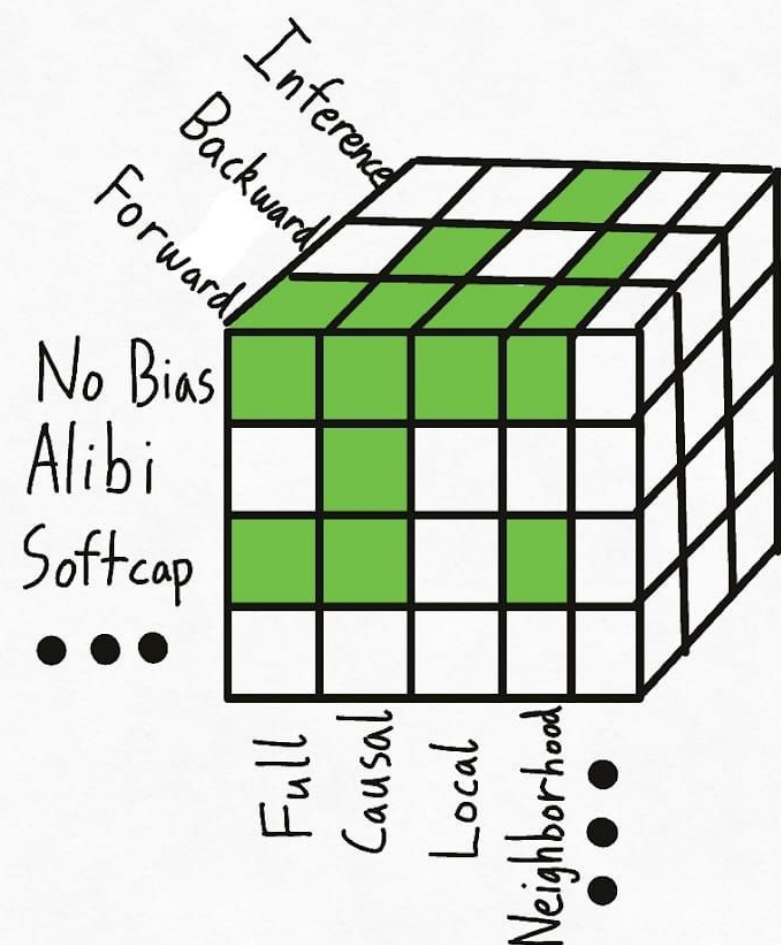
**Trivial modifications to  
the kernel that are  
codegenable**

# Abstraction: FlexAttention

## Attention Variant Support

Today

FlexAttention



# Attention vs FlexAttention

$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$

$$\text{FlexAttention}(Q, K, V) = \text{softmax} \left( \text{score\_mod} \left( \frac{QK^T}{\sqrt{d_k}} \right) \right) V$$

# Standard Full Attention (No-op)

```
def noop(score, b, h, q_idx, kv_idx):  
    return score  
  
from torch.nn.attention.flex_attention import flex_attention  
  
flex_attention(query, key, value,  
score_mod=noop).sum().backward()
```

# Alibi Bias

```
alibi_bias = generate_alibi_bias() # [num_heads]
def alibi(score, b, h, q_idx, kv_idx):
    bias = alibi_bias[h] * (q_idx - kv_idx)
    return score + bias
```



**Jonathan Frankle** 

@jefrankle



Agreed on all points. Kernels come out for RoPE first because of llama. That's why we switched to RoPE for DBRX. It wasn't better than ALiBi.

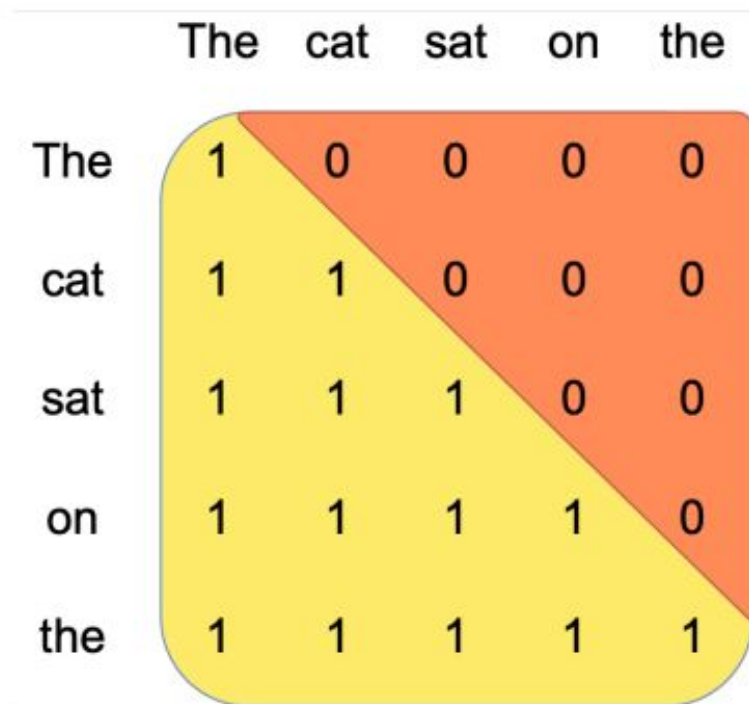
10:29 AM · Jun 22, 2024 · **1,109** Views



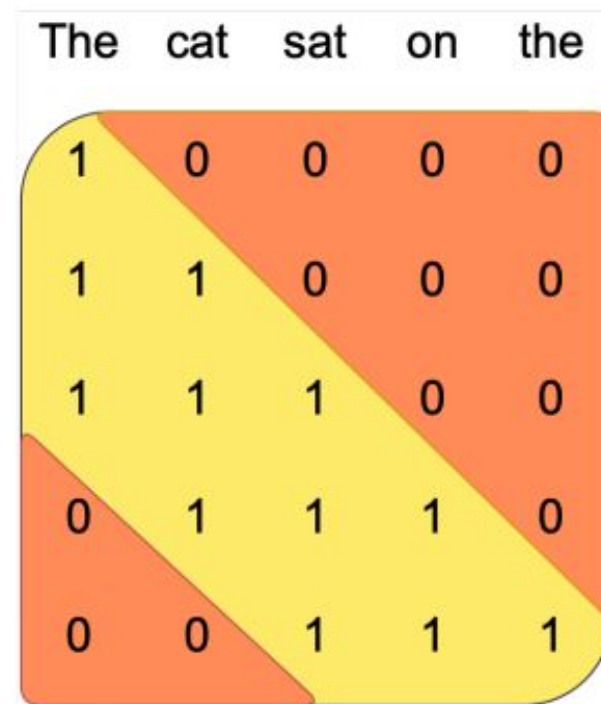
# Sliding Window Attention

```
SLIDING_WINDOW = 1024
```

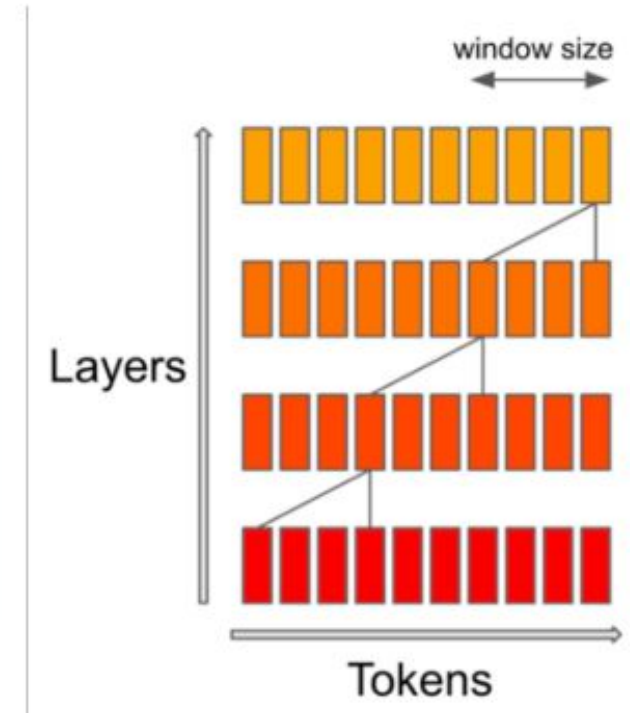
```
def sliding_window(b, h, q_idx, kv_idx):  
    causal_mask = q_idx >= kv_idx  
    window_mask = q_idx - kv_idx <= SLIDING_WINDOW  
    return causal_mask & window_mask
```



Vanilla Attention



Sliding Window Attention



Effective Context Length





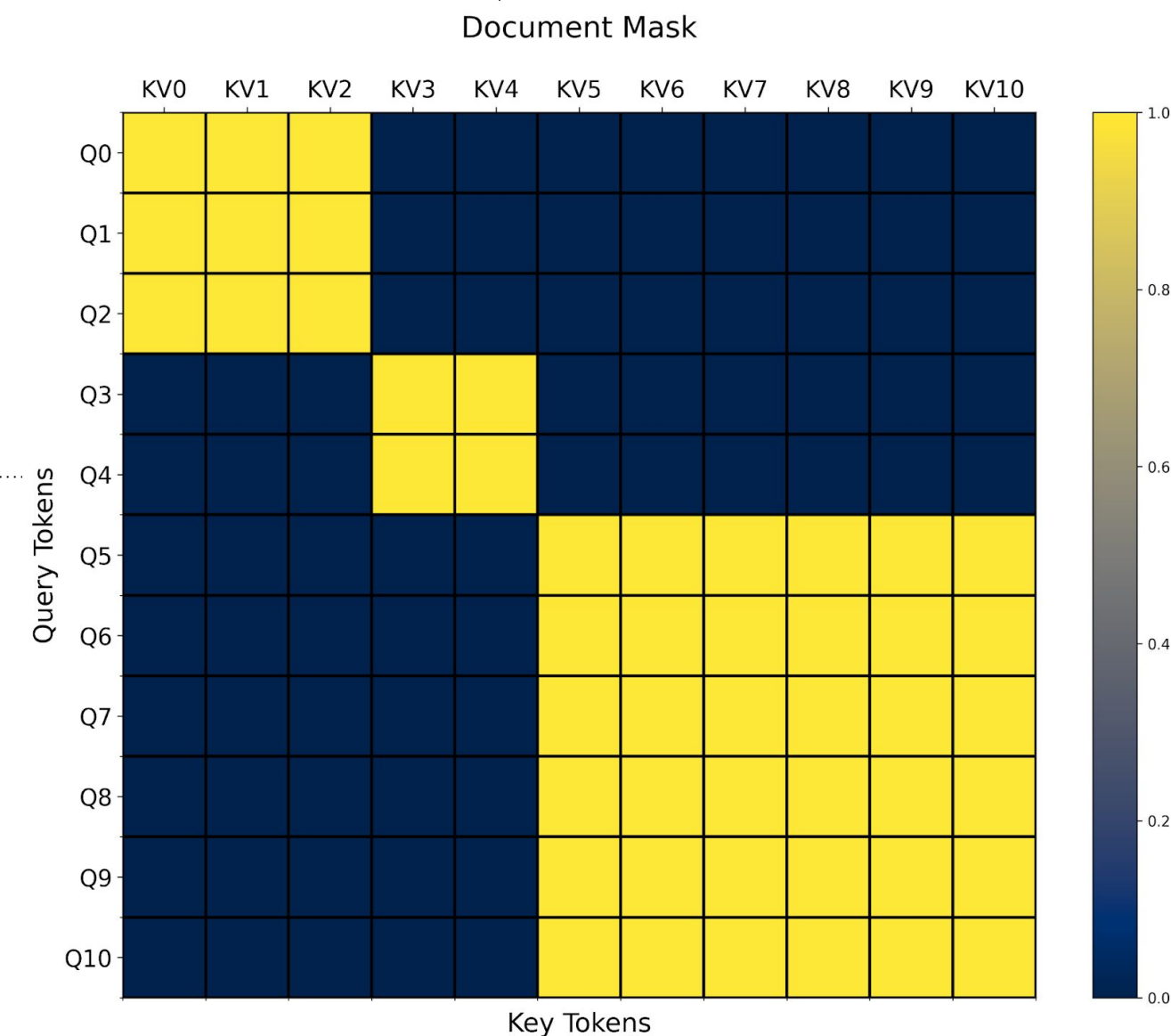
# Document Masking/Jagged Sequences

Imagine you have one sequence with 8000 tokens, and then a bunch of sequences with 2000 tokens.

```
# The document that each token belongs to  
# e.g. [0, 0, 0, 1, 1, 2, 2, 2, 2, 2, 2] corresponds to  
# sequence length 3, 2, and 6.
```

```
document_id: [SEQ_LEN]
```

```
def document_masking(b, h, q_idx, kv_idx):  
    return document_id[q_idx] == document_id[kv_idx]
```

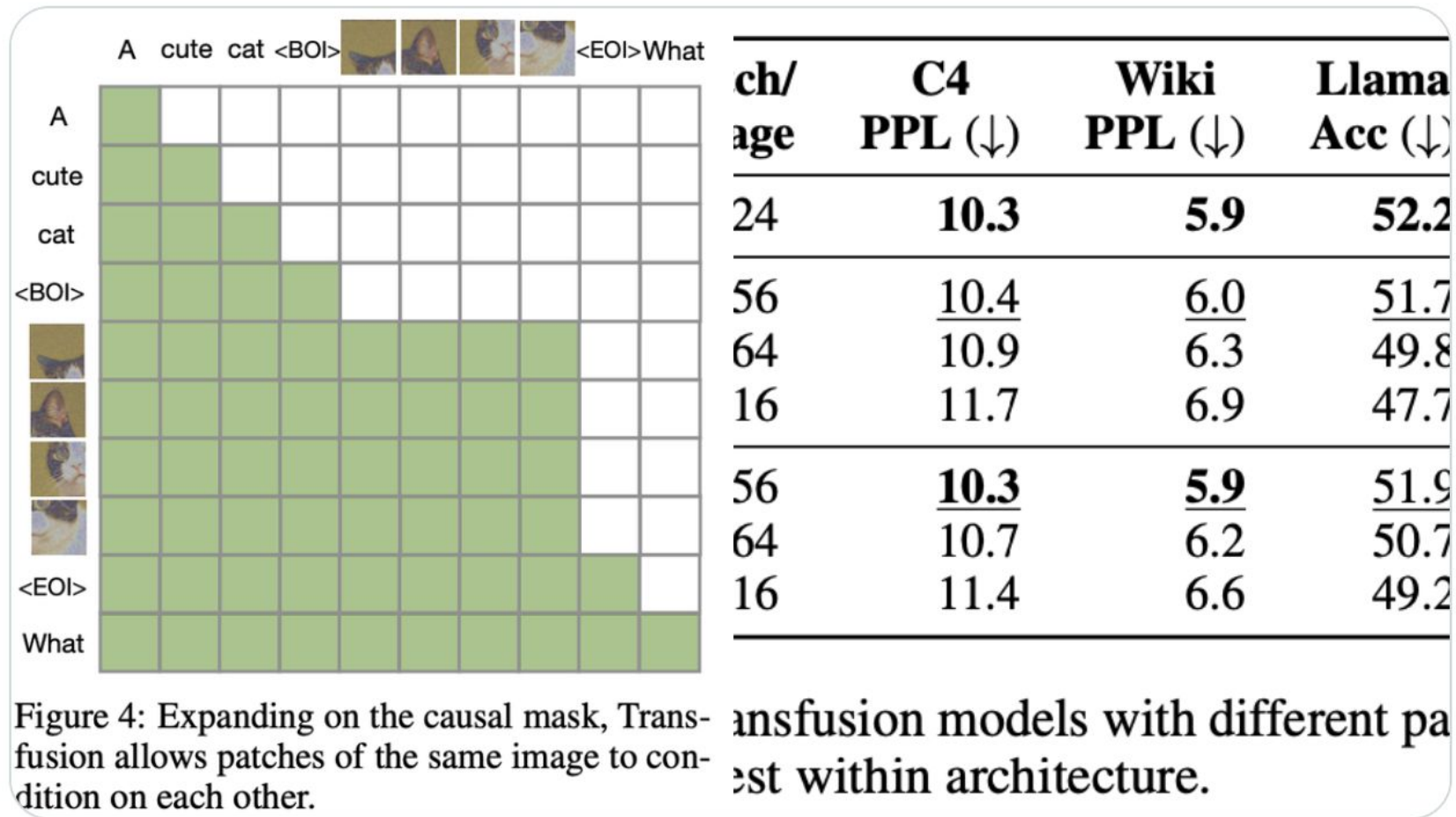




Modeling innovations:

- \* Global causal attention + bidirectional attention within each image is crucial.
- \* Introducing modality-specific encoding and decoding layers improves performance and can compress each image to 64 or even 16 patches!

4/5

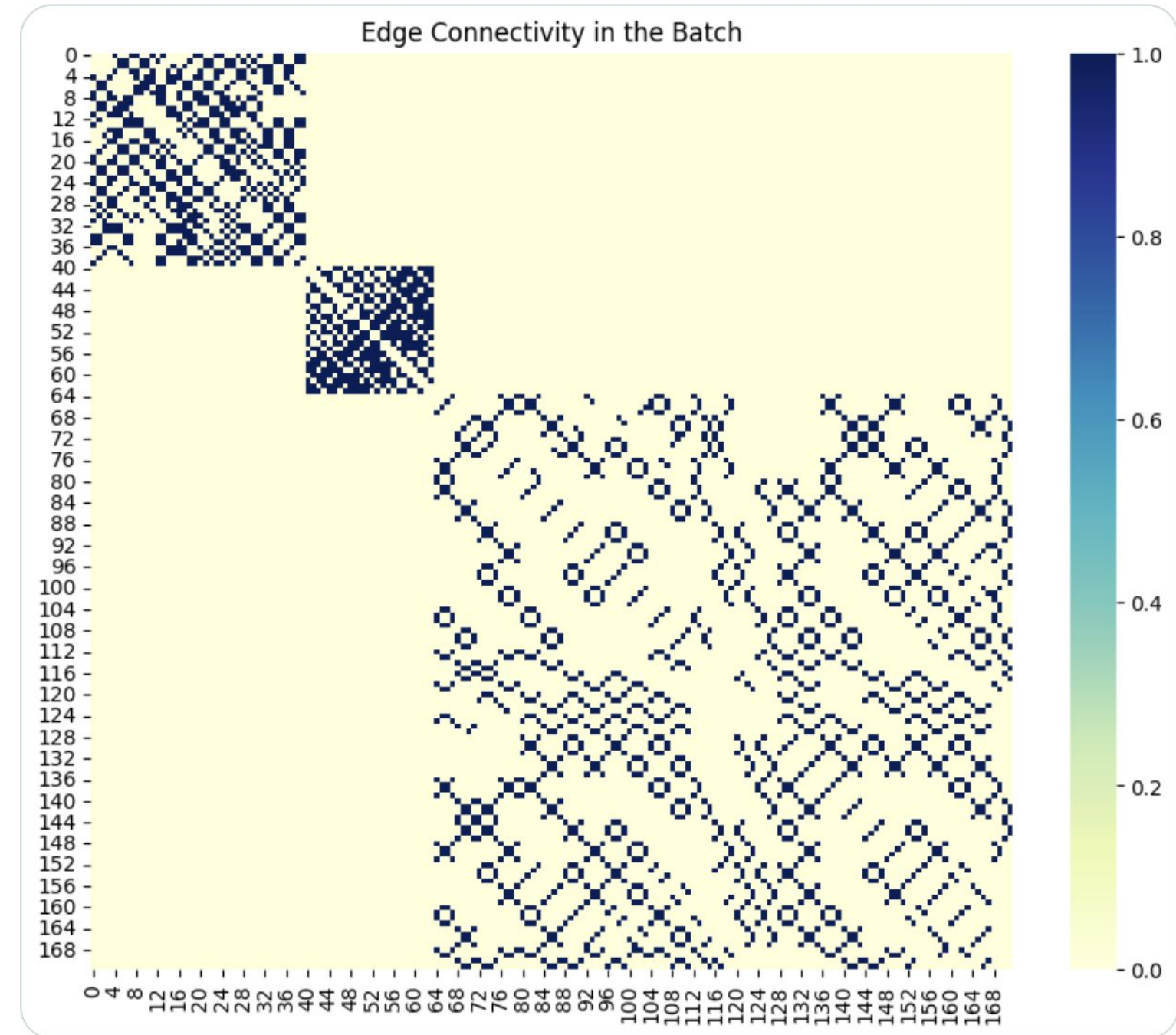


9:01 AM · Aug 21, 2024 · 25.7K Views

Pytorch Devs: Flex Attention is a beautiful computational abstraction which uses compilation and caching to create fused kernels for blocksparse attention patterns

Me: I will use it to implement this attention pattern

Pytorch Devs: .... plz

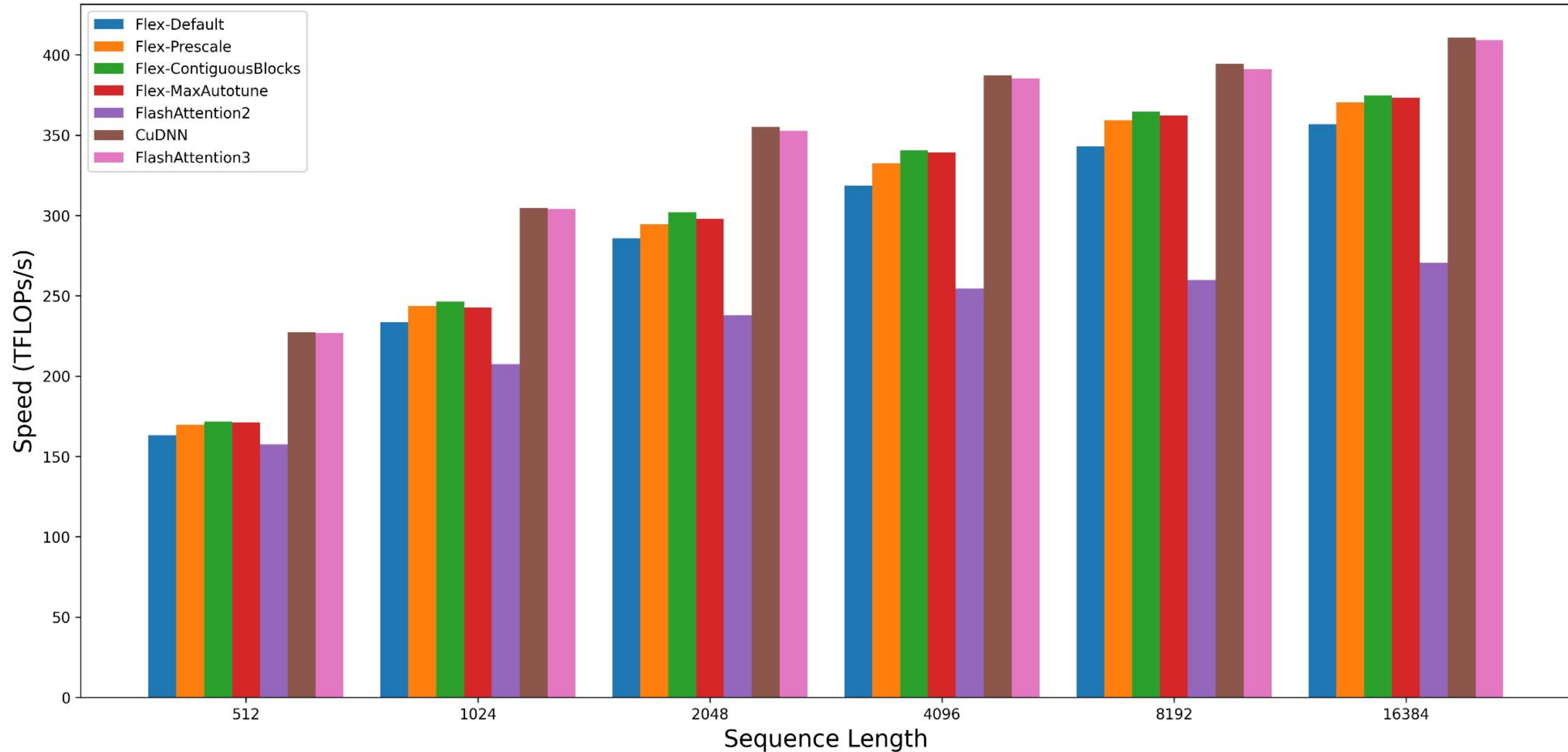


6:34 PM · Nov 1, 2024 · 25.3K Views

# Performance

## Attention Variants Comparison

Forward Pass + Causal Masking (H100)



# FlexAttention is an “abstraction”

It hides away “how do you write an efficient flashattention implementation with arbitrary masking”, but exposes “different attention variants”

It solves the problem that “writing primitive PyTorch ops is too slow/brittle”, while solving the problem that “modifying Attention kernels is a hard/involved process”

Personally, I think this is the most “interesting” kind of work in systems (coming up with the right abstractions).

# Large-Scale Training

# How do you even think about large-scale training?

DDP, FSDP, TP, PP, etc.?

But how do we choose between them?

Which one should we be using?

If our GPUs became 8x faster how would that change our parallelism strategy?

If we had to train across very-low bandwidth networking how would that change our parallelism?

**Think about the “system”, not about the technique.**

# Systems-Level Thinking for Distributed ML

Question: What are we trying to optimize?

Answer 1: We're trying to optimize the throughput of our model.

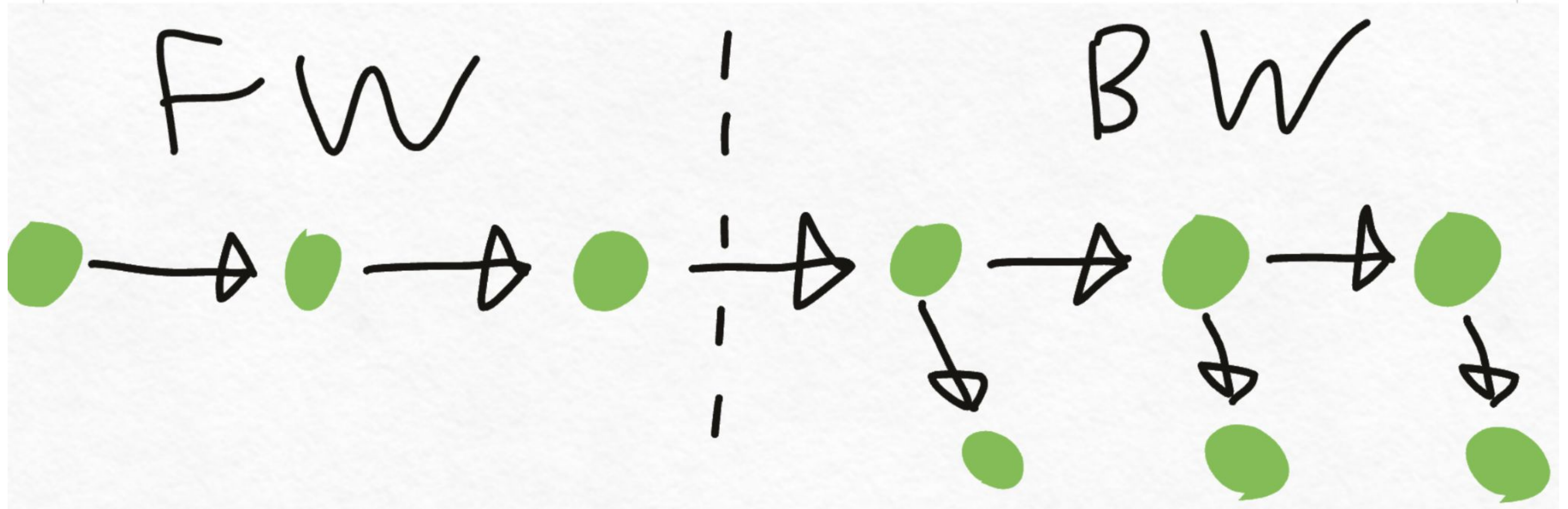
Answer 2: Given the model and batch size we're training with, optimize the throughput.

Answer 3: Given the model and batch size we're training with, maximize the amount of time we're doing matrix multiplications, and minimize the time spent

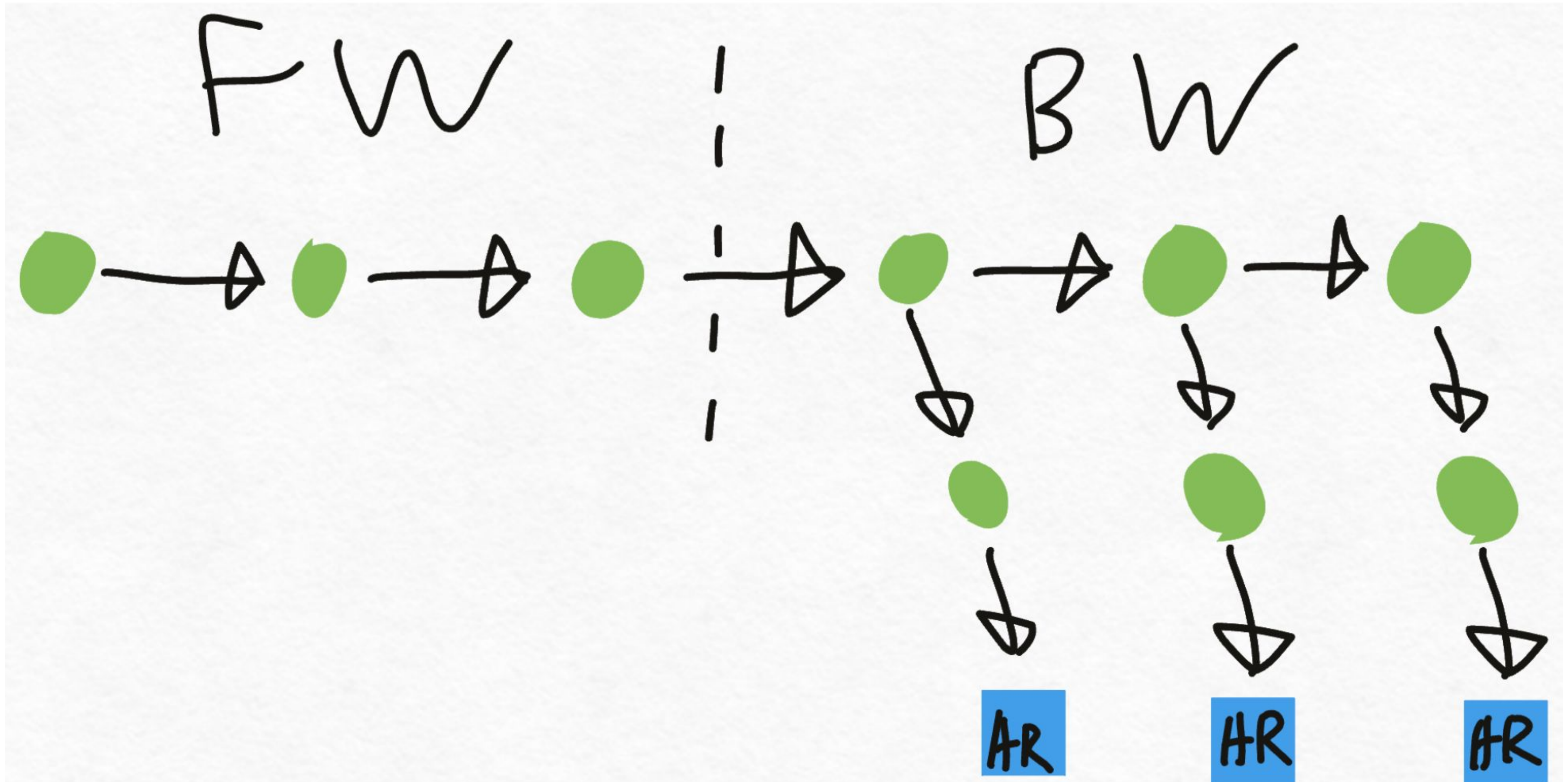
We're trying to come up with some "abstraction" to make our life easier.



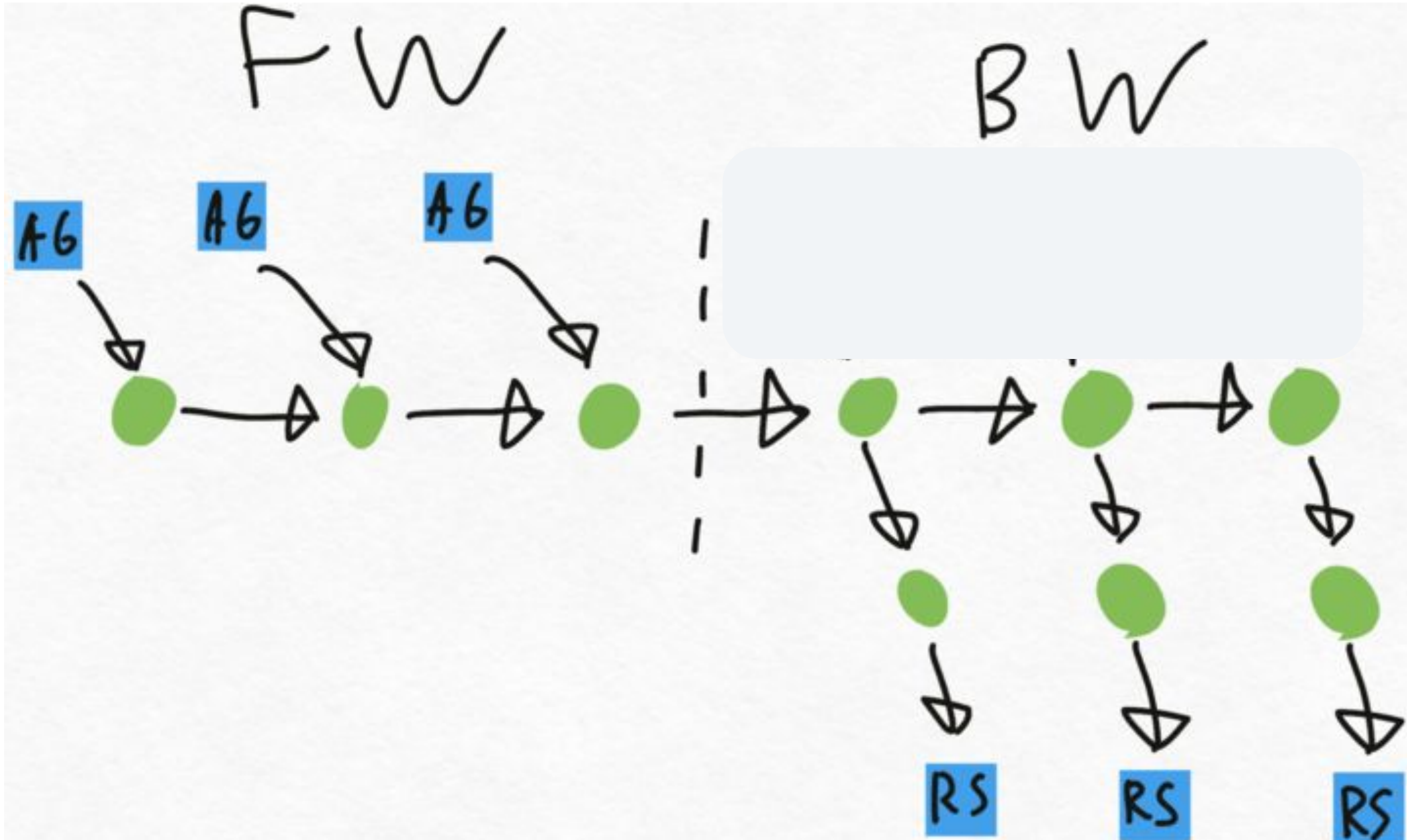
# Normal Forwards and Backwards



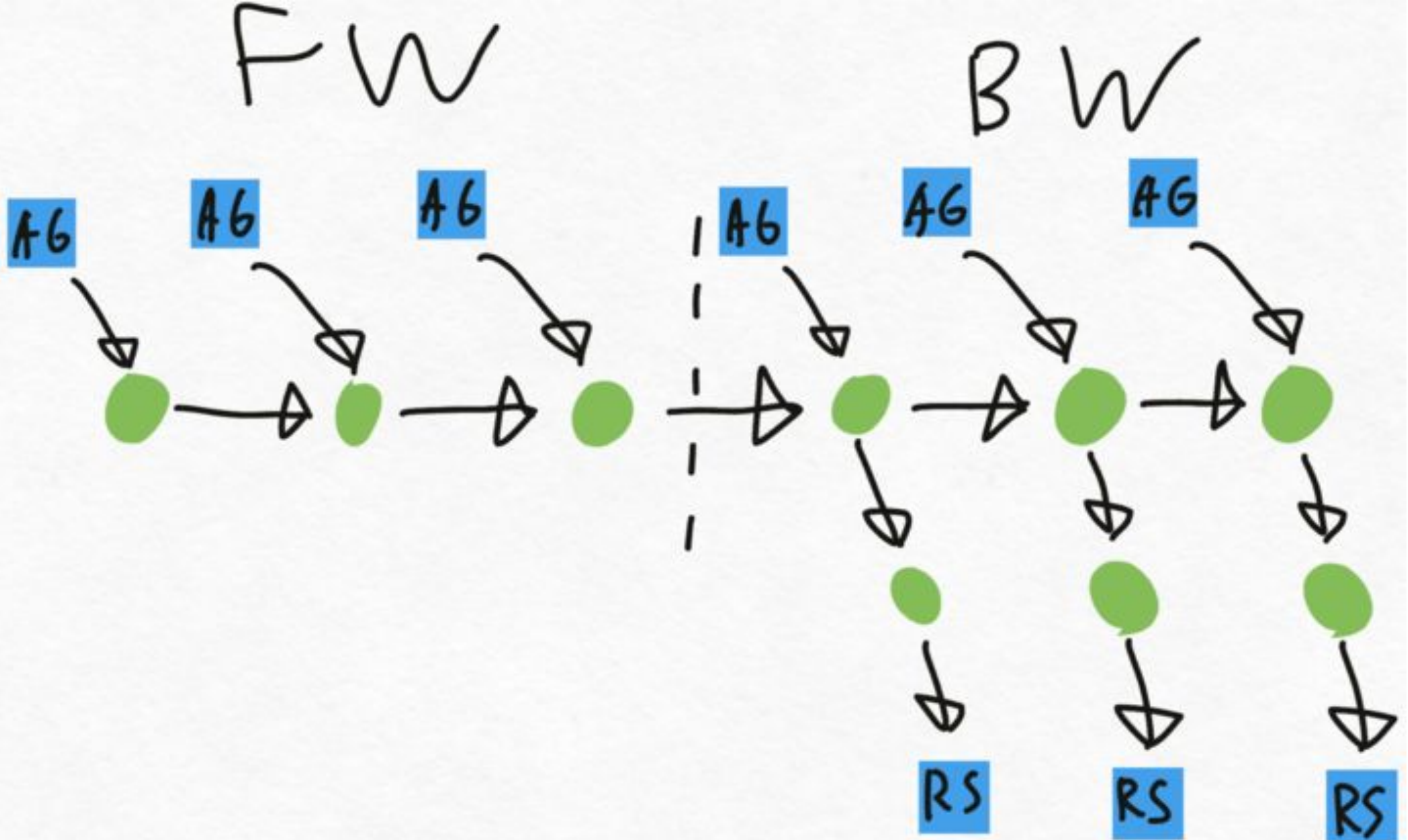
# DDP



# Zero-1/Optimizer State Sharding



# Zero-3/Fully Sharded Data Parallel



# Why can't we just only use FSDP?

**Observation 1: If the total computation time exceeds FSDP's total communication time, then FSDP's communication can be fully overlapped and is considered to be "free".**

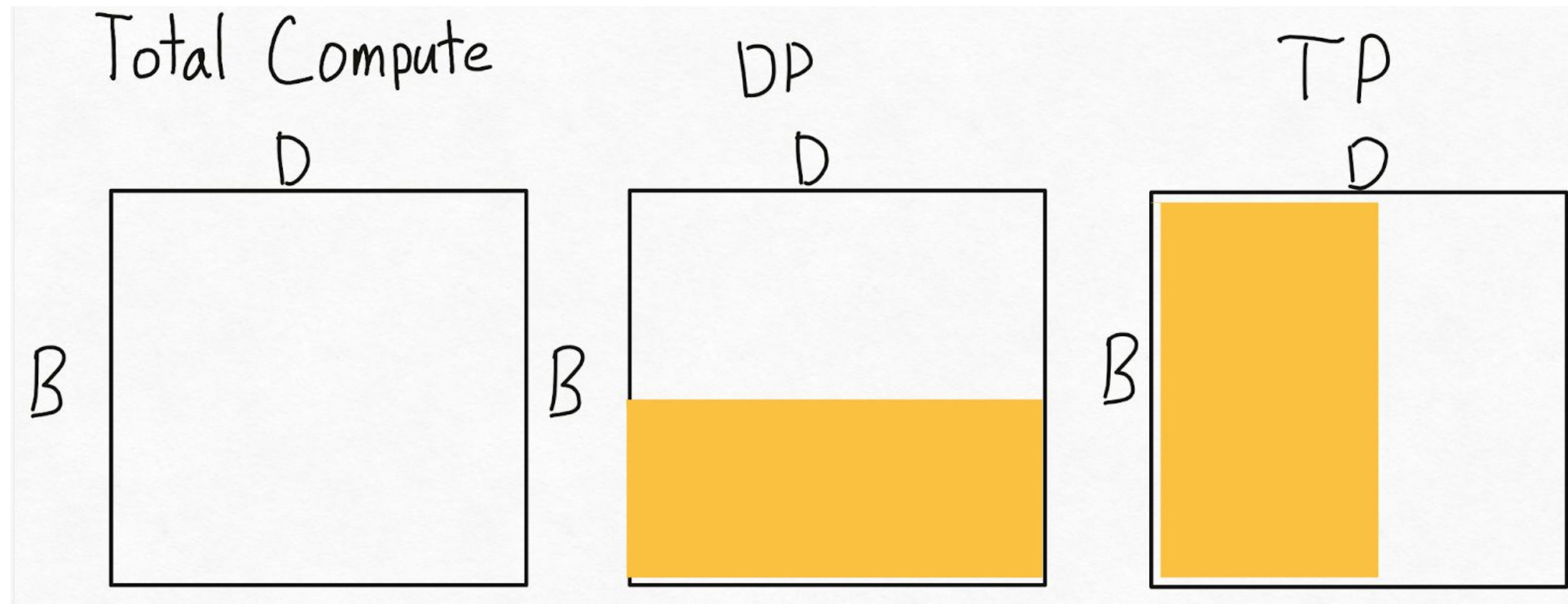
## **When we can use FSDP**

**Compute Time > Communication Time**

# Assumptions

1. The global batch size we're using is identical
2. The model architecture is identical

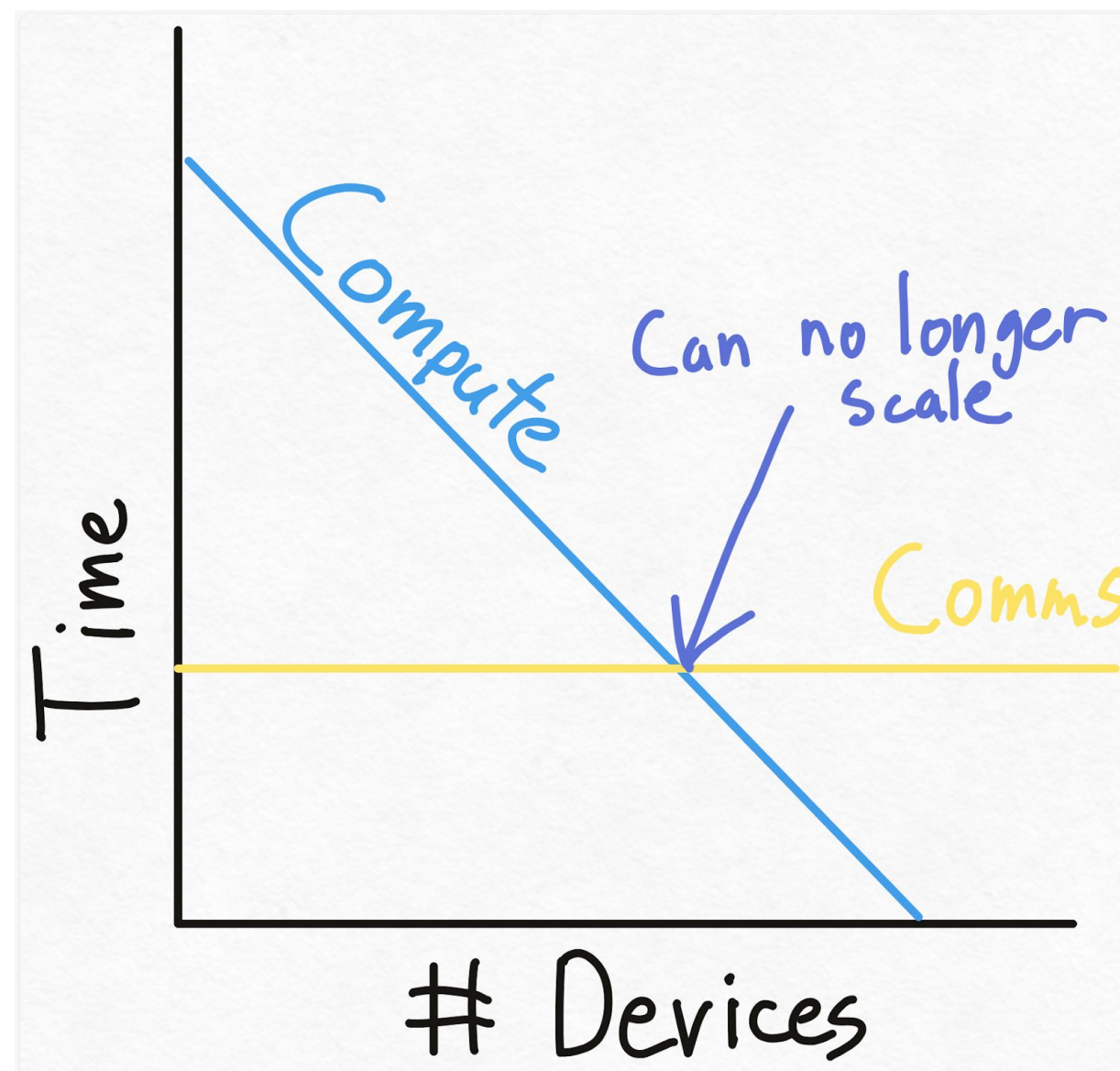
Therefore... **Observation 2: How long we spend on compute is independent of our parallelism strategy\***



# What happens to compute and comms when we double GPUs?

Compute: Doubling our number of devices halves how long our compute takes

Comms: Our FSDP comms don't get any faster as we scale how many devices we have\*



# Concrete Numbers on Scaling with FSDP

FSDP comms time == Computation time

FSDP comms time

= 70 billion [parameters] \* 2 [bytes per param] \* 3 [collectives (2 allgathers and a reducescatter)] / 200 billion [Gigabytes/second internode bandwidth]

= 70 billion \* 2 \* 3/100 billion

= 2.1 seconds

Computation time

= (70 billion [parameters] \* 6 [flops per param per token] \* 4 million [total tokens]) / 200 trillion [teraflops of fp16 compute] / **4000 [GPUs]**

= 70 billion \* 6 \* 8 million / 200 trillion / 4000

= 2.1 seconds



# Local Batch Size Constraint

If we have 8 million tokens global batch size, and 8k sequence length, then we only have 1000 sequences!

Since FSDP requires one sequence per GPU, we can't scale over 1000 GPUs.

# If you're scaling FSDP, there are 4 situations

Case 1: You haven't hit either the local batch size limit or the FSDP comms limit.

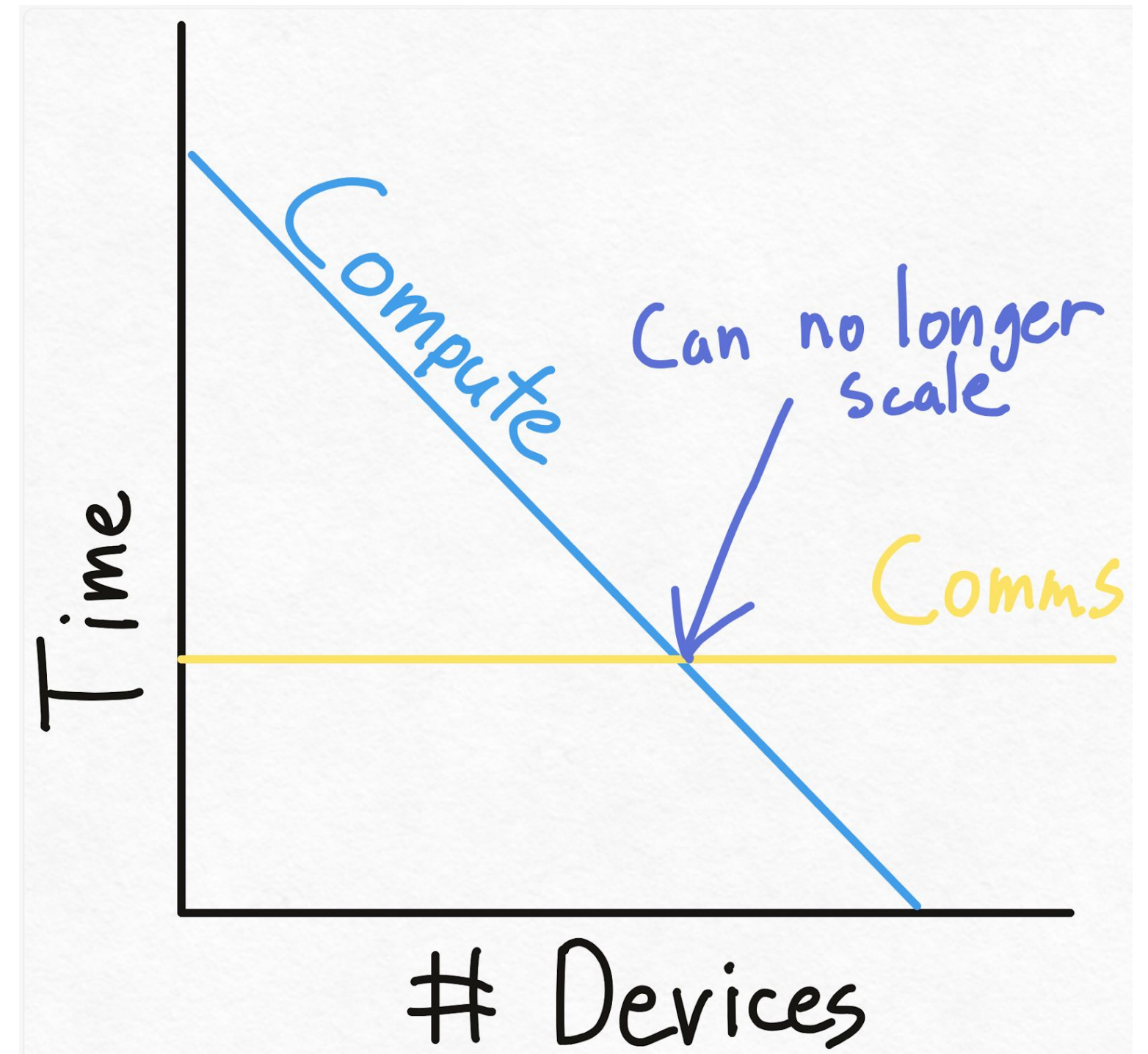
Case 2: You've hit the local batch size limit, but not the FSDP comms limit.

Case 3: You've hit the FSDP comms limit, but not the local batch size limit.

Case 4: You've hit the FSDP comms limit and the local batch size limit.

Case 1: You haven't hit either the local batch size limit or the FSDP comms limit.

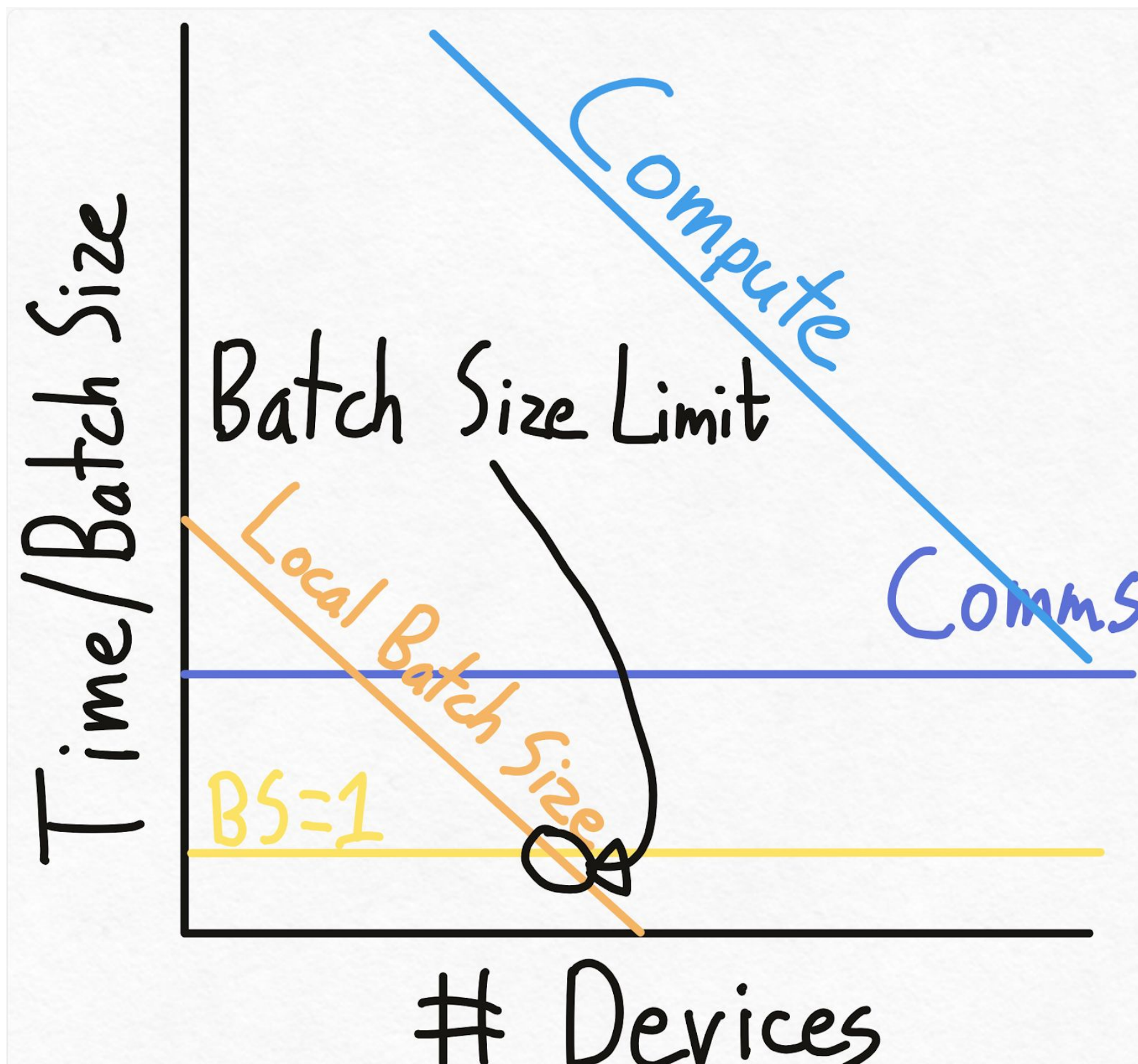
Yay! Keep using FSDP



# Case 2: You've hit the local batch size limit, but not the FSDP comms limit.

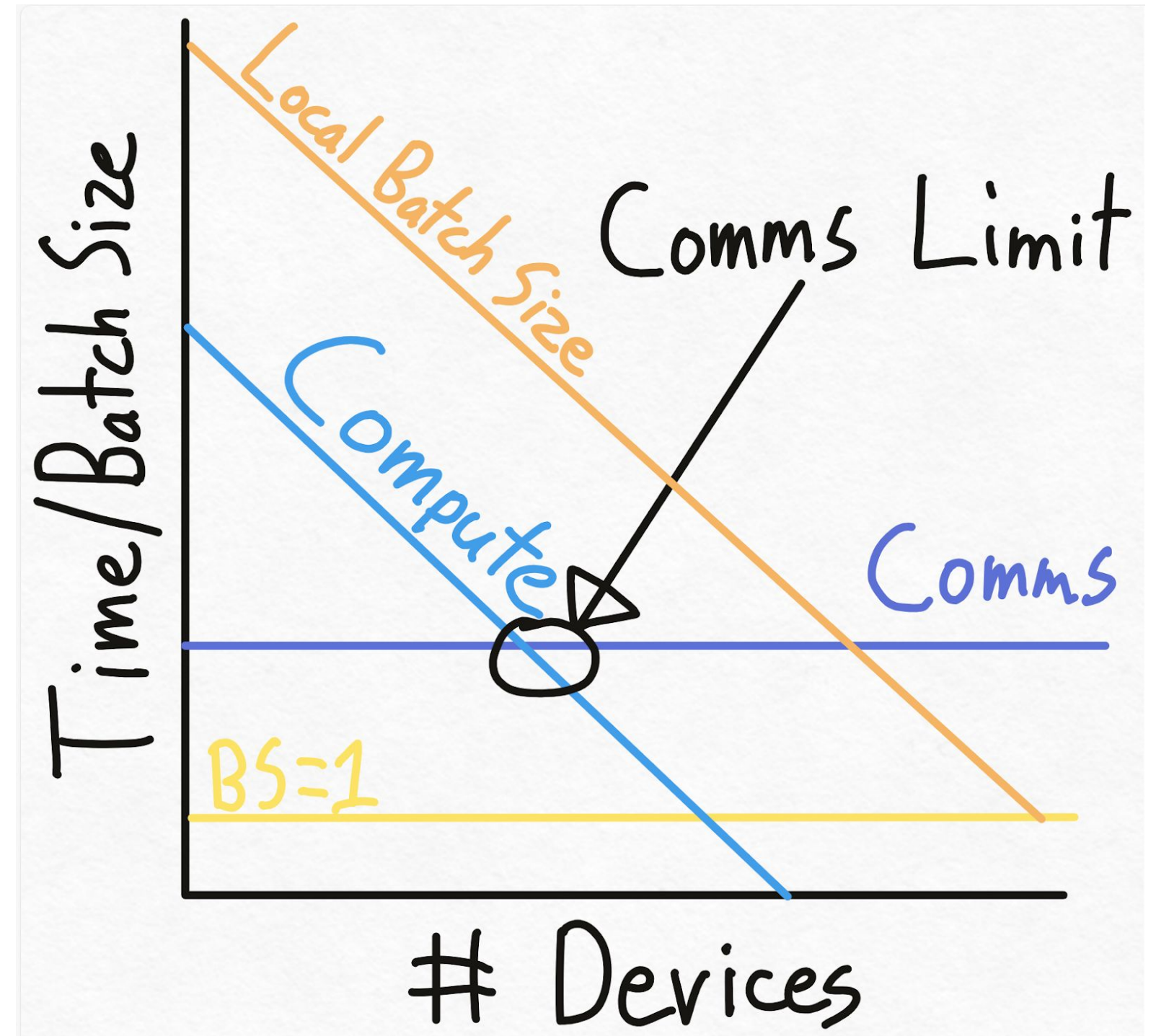
Solution: Use CP or TP.

Note that PP does not help!



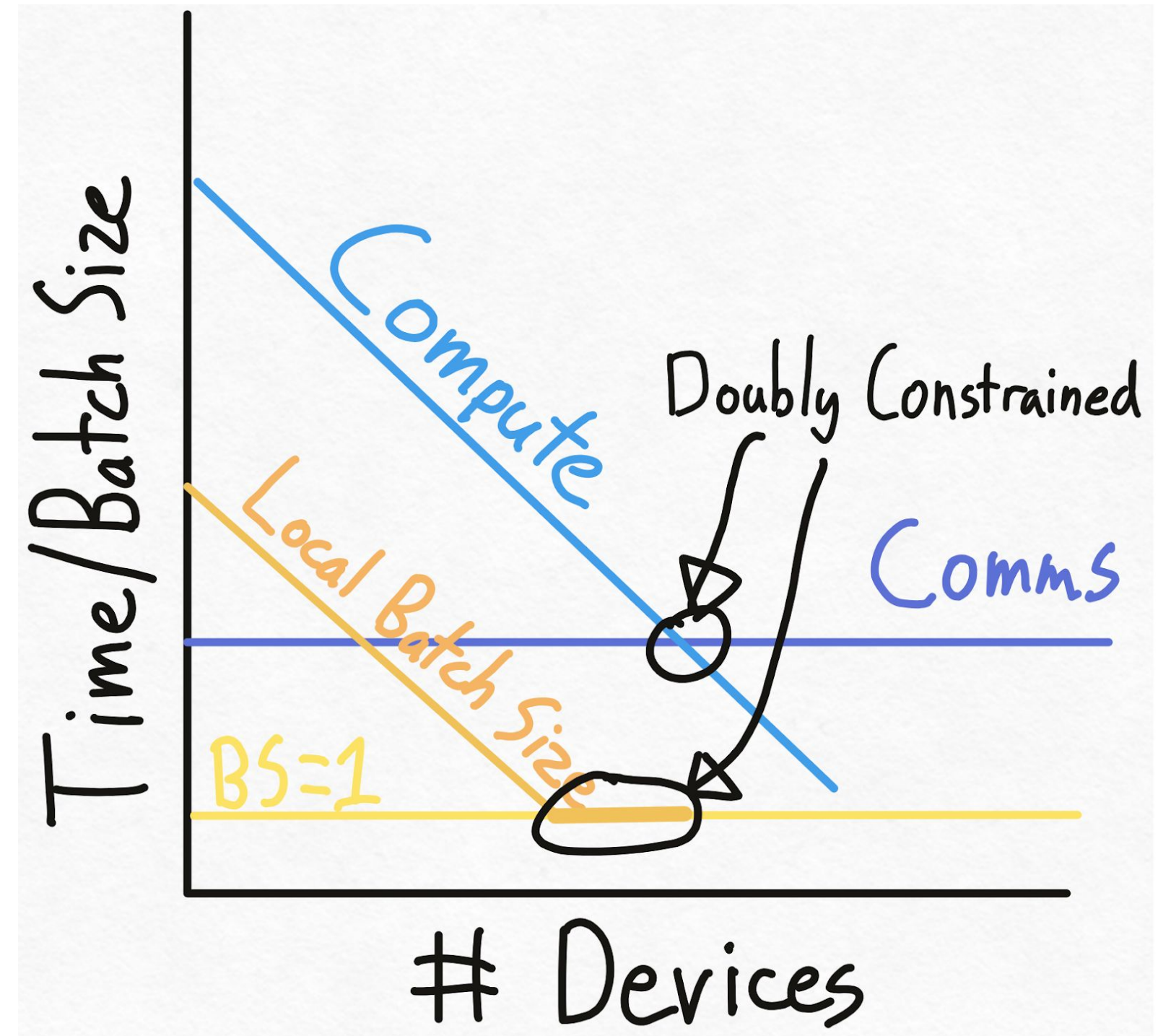
# Case 3: You've hit the FSDP comms limit, but not the local batch size limit.

Solution: Use PP (or inter-node TP)



# Case 4: You've hit the FSDP comms limit and the local batch size limit.

Solution: Use CP/TP + PP or inter-node TP



# In our setting, how does increasing PP effect the bubble size?

A: Significant Increase

B: Slight Increase/No Change

C: Significant Decrease

## Assumptions

1. The global batch size we're using is identical
2. The model architecture is identical

<https://claude.site/artifacts/06c22826-52fa-4210-8869-824f43b9eba5>

# Abstraction Leak: GPUs don't error(?)

10 Failures per 1000 node-days: <https://arxiv.org/pdf/2410.21680v1>

With 1 node, that's one failure per 100 days.

With 100 nodes, that's one failure per day.

With 10000 nodes (80k GPUs), that's one failure per 15 minutes.

Uh oh....



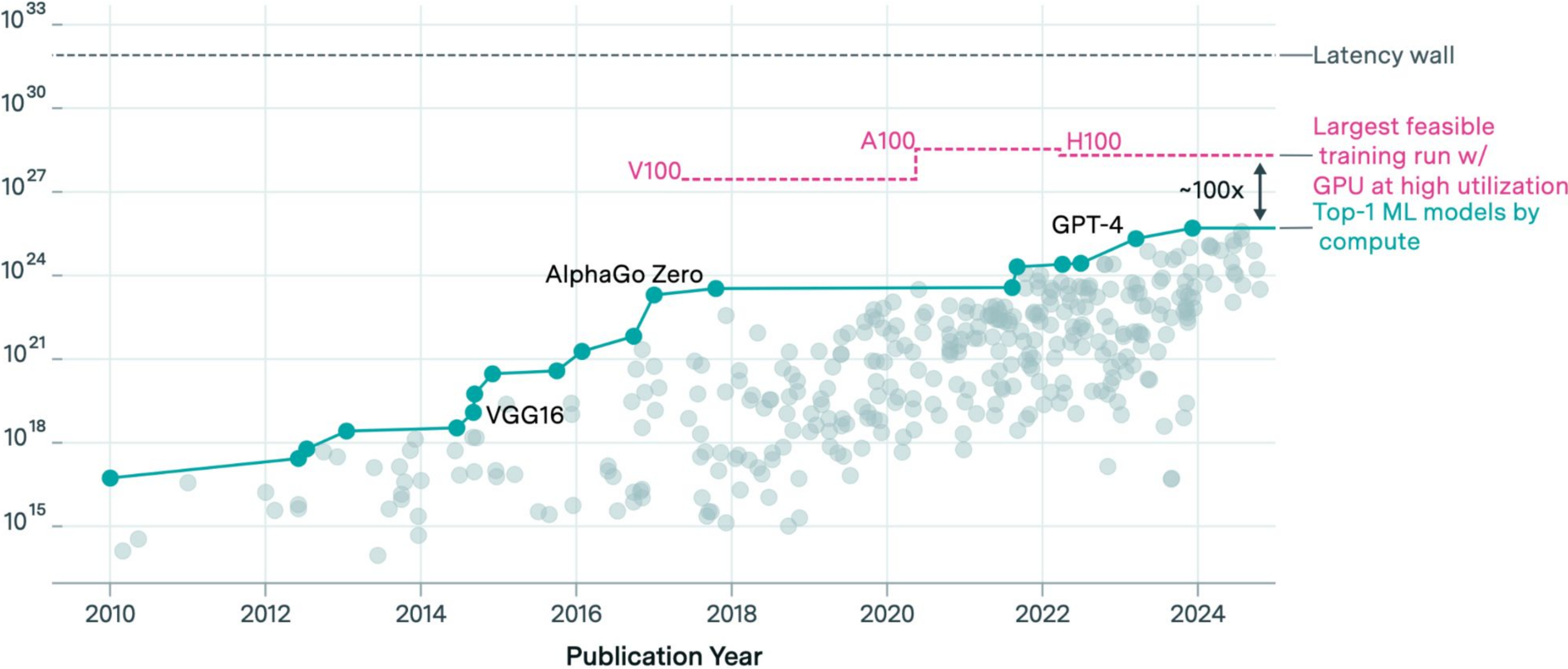
# Fault Tolerant Training

If a GPU dies, instead of killing the whole job, let's just drop the “replica” that that GPU is part of.

This leads to us running with a (reduced) global batch size (not semantics preserving).

# Latency “Wall”

Training compute (FLOP)



# Diloco (semi-sync training)

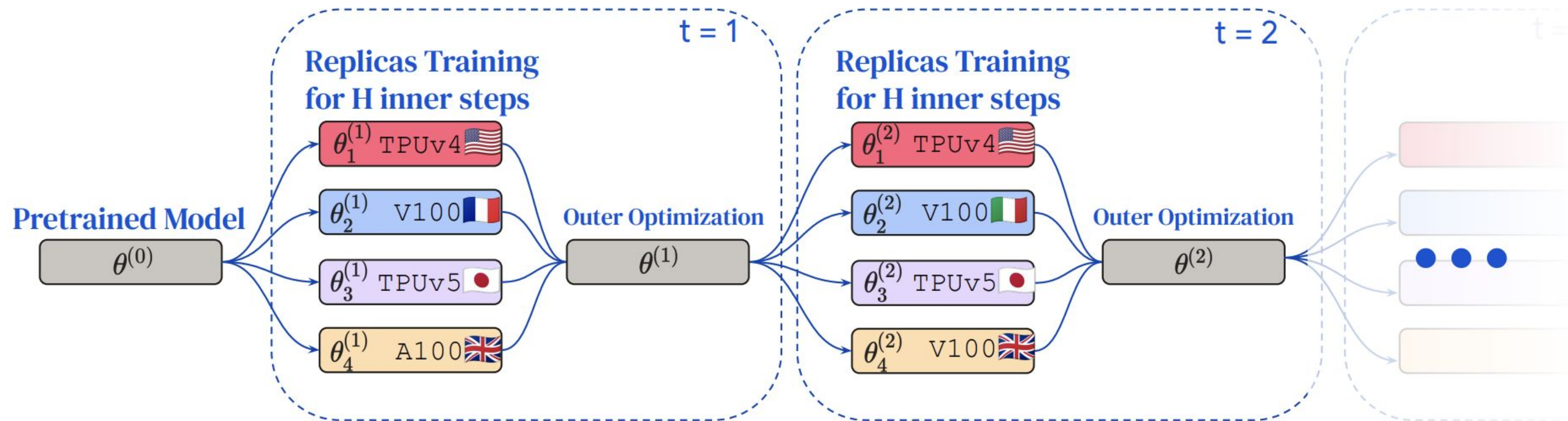


Figure 1 | **DiLoCo**: First, a pretrained model  $\theta^{(0)}$  is replicated  $k$  times (in this illustration  $k = 4$ ) and each worker  $\theta_i^{(1)}$  trains a model replica on its own shard of data for  $H$  steps independently and in parallel. Afterwards, workers average their outer gradients and an outer optimizer updates the global copy of the parameters  $\theta^{(1)}$ . This will then be re-dispatched to the workers. The process repeats  $T$  times (in this illustration only the first two iterations are displayed). Each replica can be trained in different locations of the world, with different accelerators.

# Interesting Future Questions

1. How can we modify our neural network architectures to be more fault tolerant?
2. Can we improve our optimizers to deal better with our communication requirements?

Thanks for listening!

# Bonus things I can talk about

Async-TP

Zero-Bubble Pipeline Parallelism

Context Parallelism