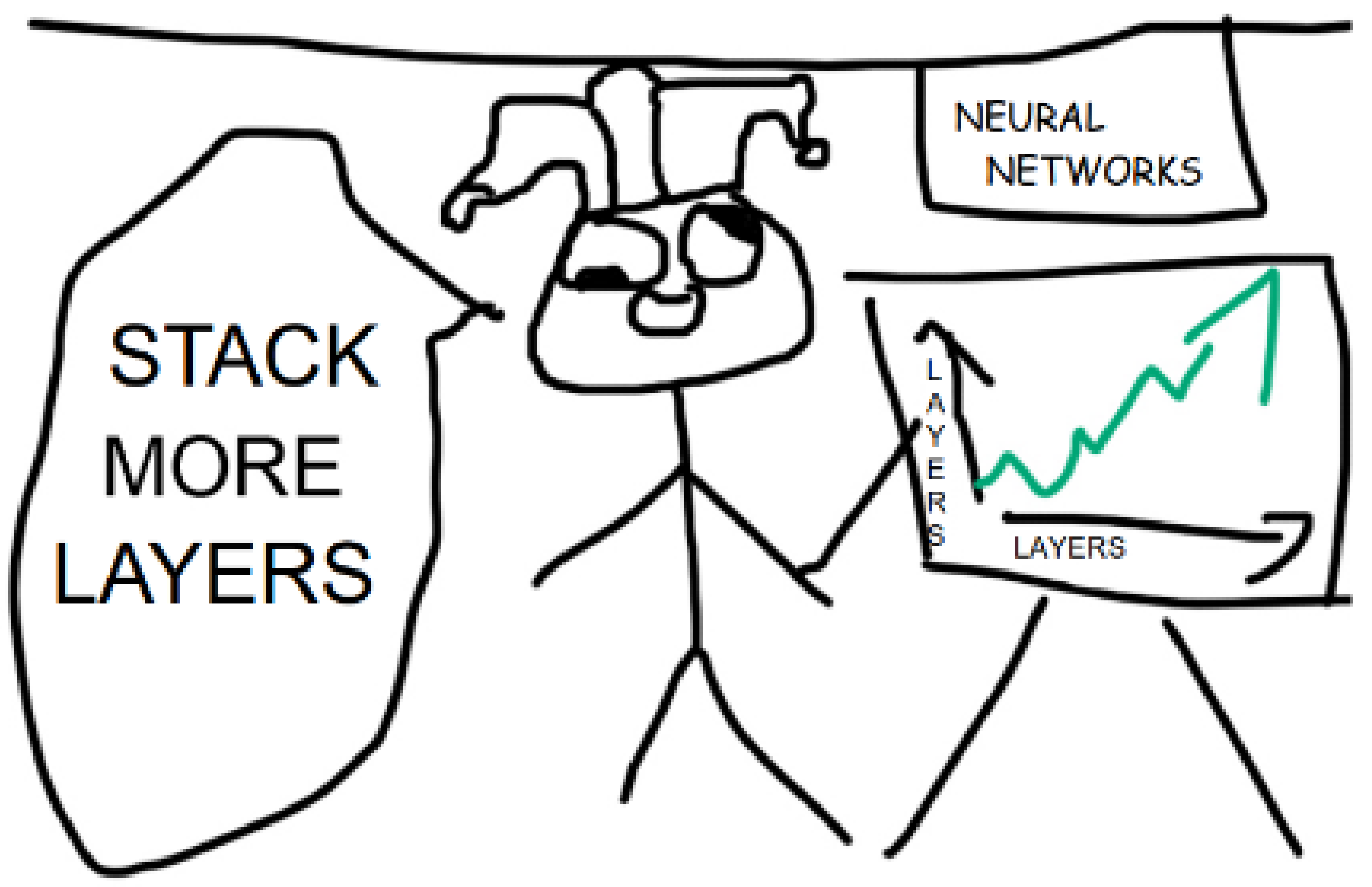
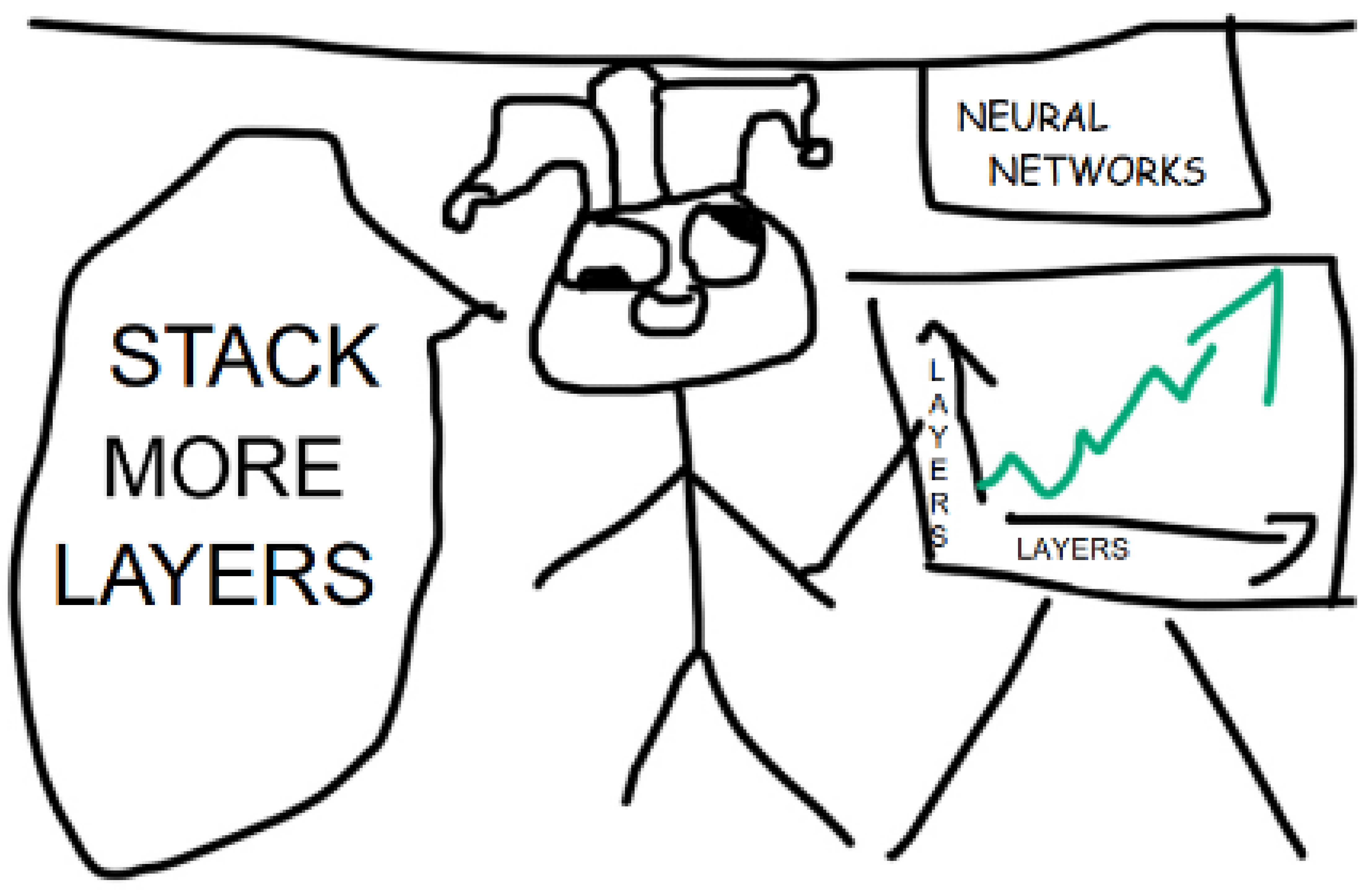
Building Machine Learning Systems for the Age of Really Big Models

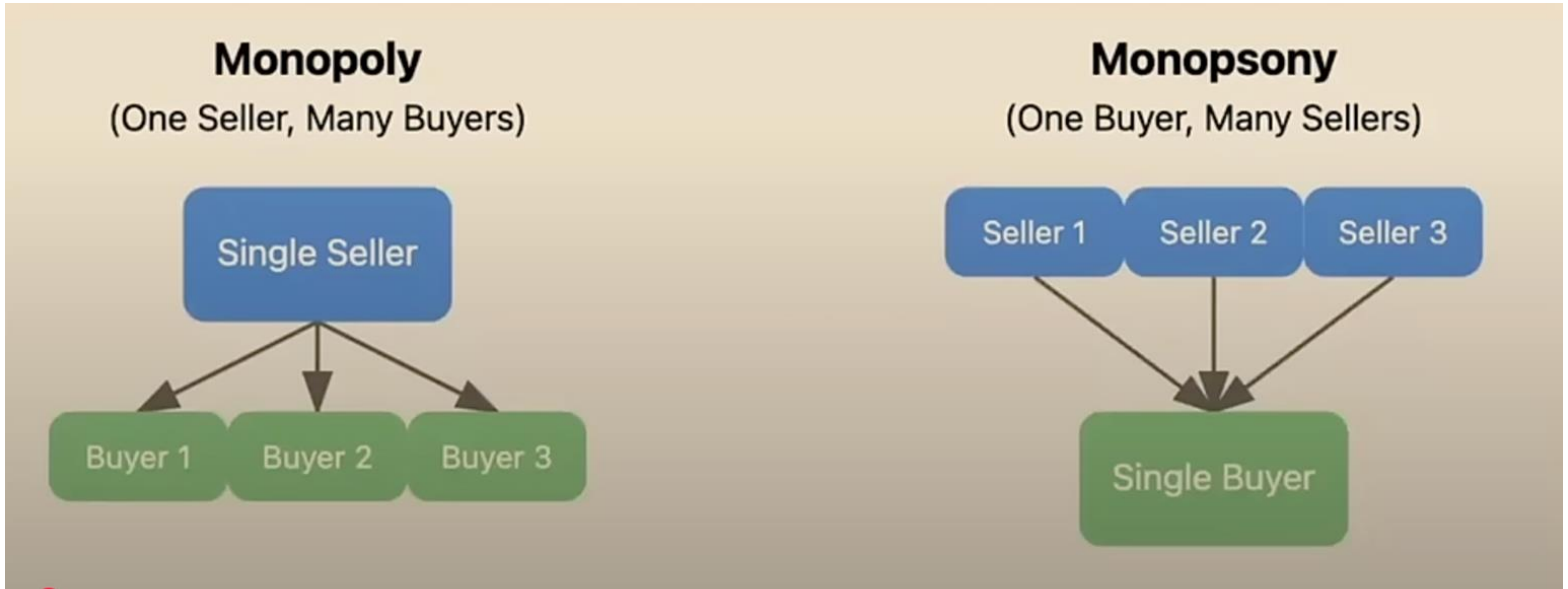


Horace He



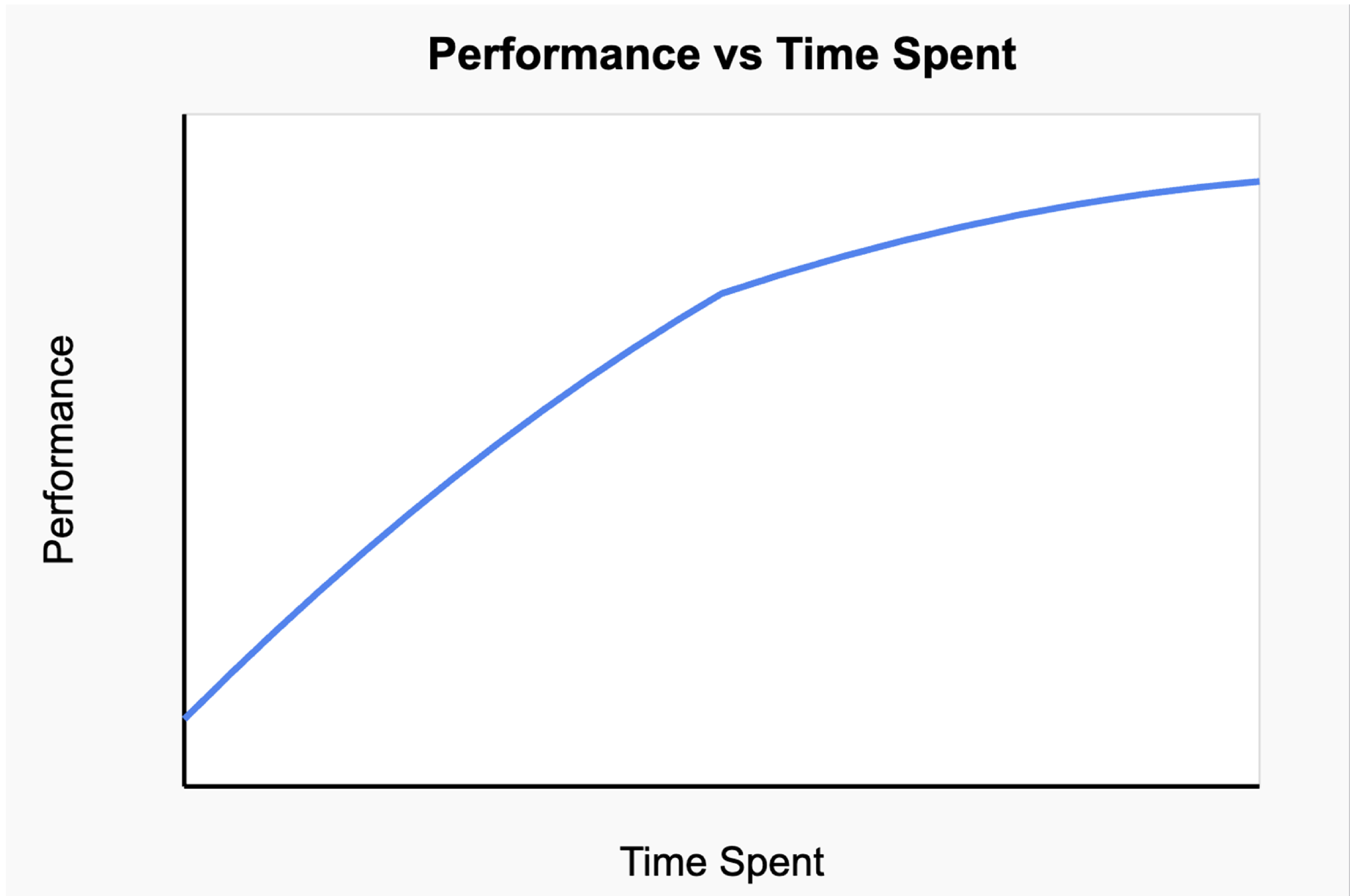
The field has consolidated significantly

- “Many architectures” => “one” architecture (transformers)
- “Many folks training SOTA models” => a few companies training SOTA models



Different programming models are useful for different folks

- It doesn’t make that much sense to ask “is torch.compile faster than CUDA”?
- What might be useful if you’re just doing experiments might not be useful if you’re about to do a 1 million GPU run
- There is a fundamental tradeoff between “more control” and “less control”



Why can’t a graph compiler do everything for me?

- Programming model is simple - put graph of operators in, get fast performance out!
- It’s the programming model that TorchInductor/XLA/tinygrad/many others are based around.
- It allows you to leverage performance improvements by experts for all users!
- They’re successful (and useful) in many cases, but folks often find them frustrating and difficult to use (especially at the frontier).

Introducing Horace’s Exciting Library (HEL)

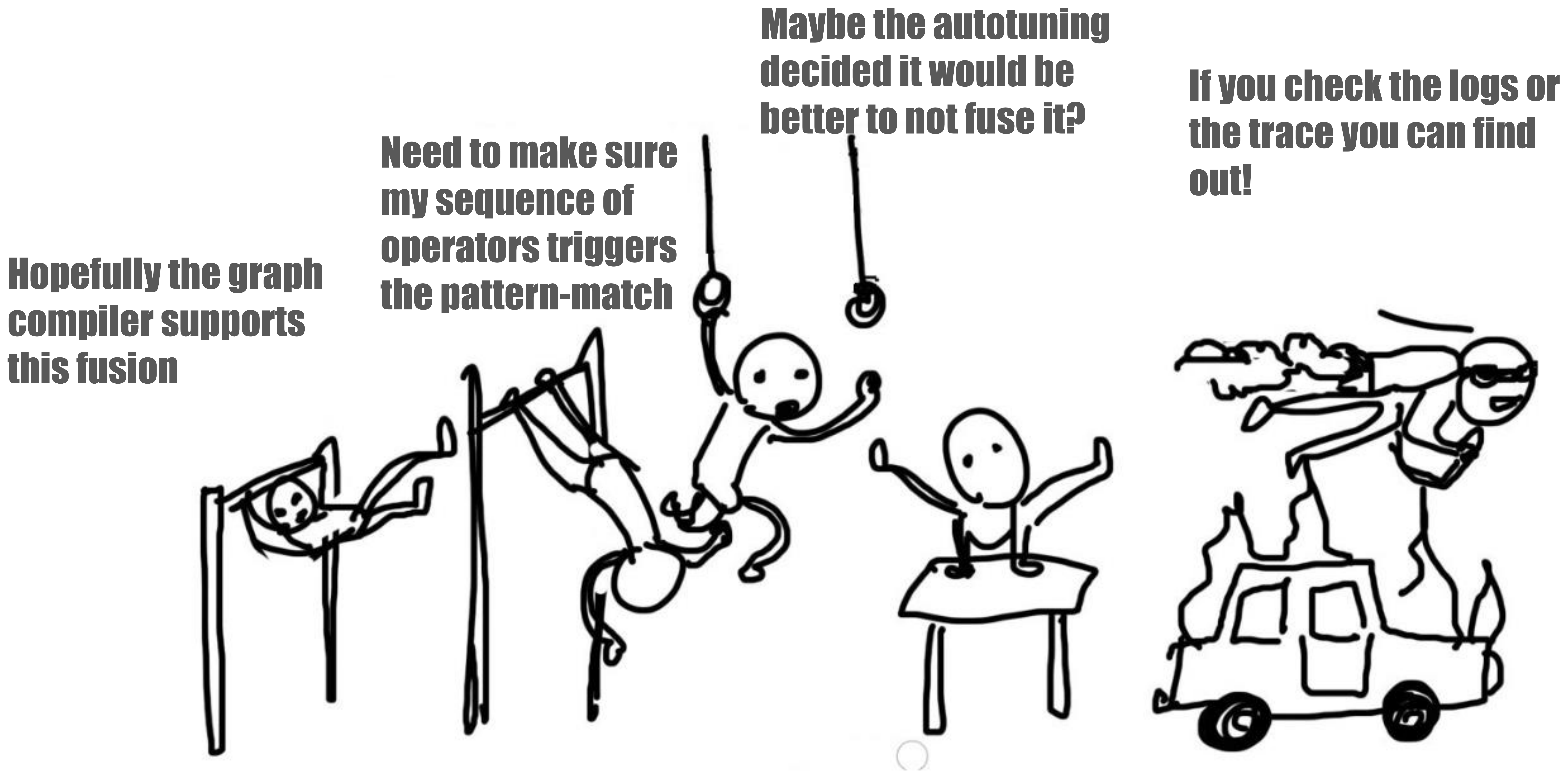
It has a couple of cool features:

1. It doesn’t always work.
2. To address that, it has no documentation on when it will work except by DM’ing me on Twitter or reading my code.
3. In exchange, when you update the library, it may totally change what code works and what code doesn’t work (i.e. no backwards compatibility)

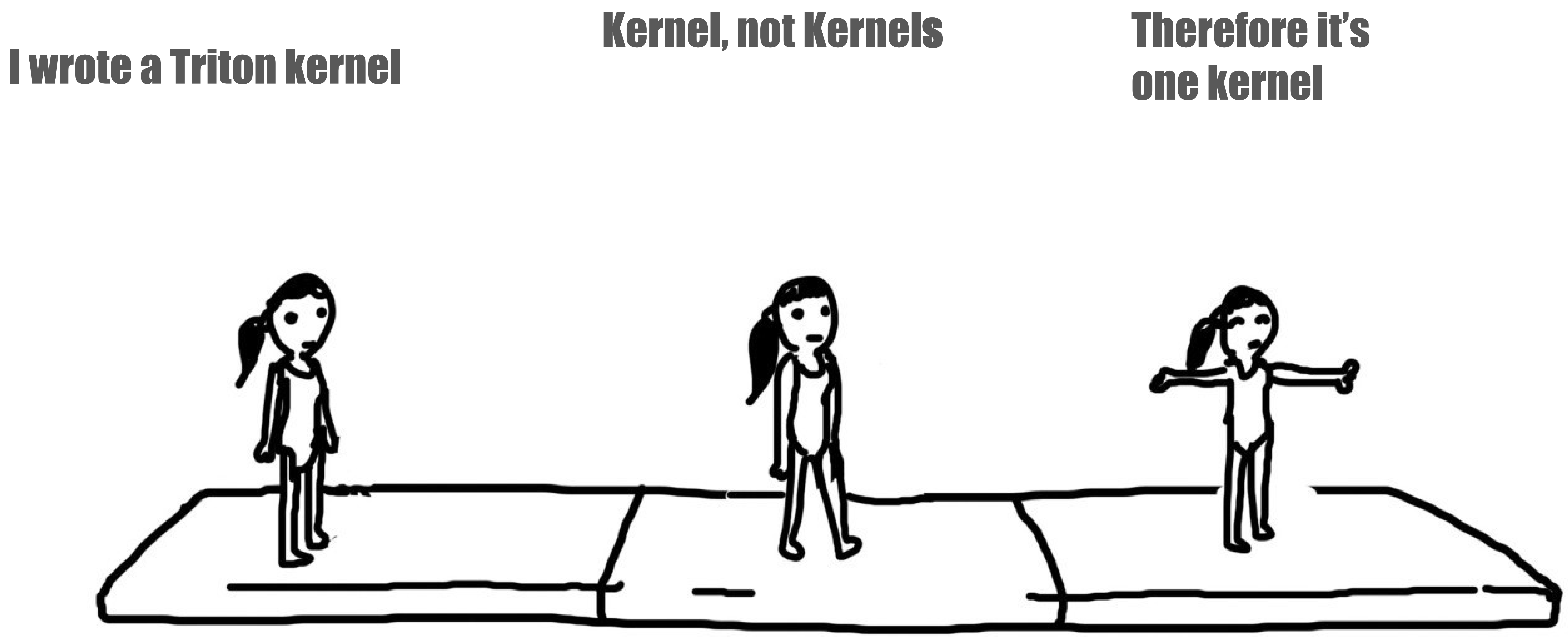
Are you interested in HEL?

Assuming that “work” = “it does my desired fusion”, this is describing graph compilers!

Asking Graph Compiler: “Will this function be fused into one kernel?”


Hopefully the graph compiler supports this fusion
Need to make sure my sequence of operators triggers the pattern-match
Maybe the autotuning decided it would be better to not fuse it?
If you check the logs or the trace you can find out!

Asking Triton: “Will this function be fused into one kernel?”



Performance is one thing, but “correctness” another

1. A bug was found where softmax gave significantly different results under compilation vs. without.
2. It was determined to only repro on CPUs but not GPUs.
3. It was determined that it only started reproing after v18 - prior to v18 it worked fine.
4. It was tracked down to <hardware vendor> adding a rewrite pass that pattern-matched softmax to their special kernel (with subtly different numeric properties).
5. However, v19 of the library actually *broke* the hardware vendor’s pattern-matching through a different way of writing softmax, causing the bug to be “fixed”!
6. Perhaps in v20 the hardware vendor would refix the pattern-matching, causing the bug to show up again.

Auto-vectorization is not a programming model

I think that the fatal flaw with the approach the compiler team was trying to make work was best diagnosed by T. Foley, who's full of great insights about this stuff: *auto-vectorization 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.

Sufficiently Smart Compiler

This is a classic argument often pulled out in a LanguagePissingMatch. The gist is that the HighLevelLanguage **H** may be slower than the LowLevelLanguage **L**, but given a SufficientlySmartCompiler this would not be the case. Moreover, this hypothetical compiler could use the high-level information available in language **H** to perform optimizing transformations which are simply not possible in **L**, thereby making an optimally-compiled **H** even *faster* than an optimally-compiled **L**.

In reality, there are a few such compilers, such as the Stalin compiler for SchemeLanguage (which was never intended to be usable on huge, production-level programs), but that's not the point. It's mostly an empty argument that's trotted out whenever a language is put down for not being AsFastAsCee.

Find the right programming model for the right user!

Programming Model: The user must do X, and then we guarantee Y but hide away Z.

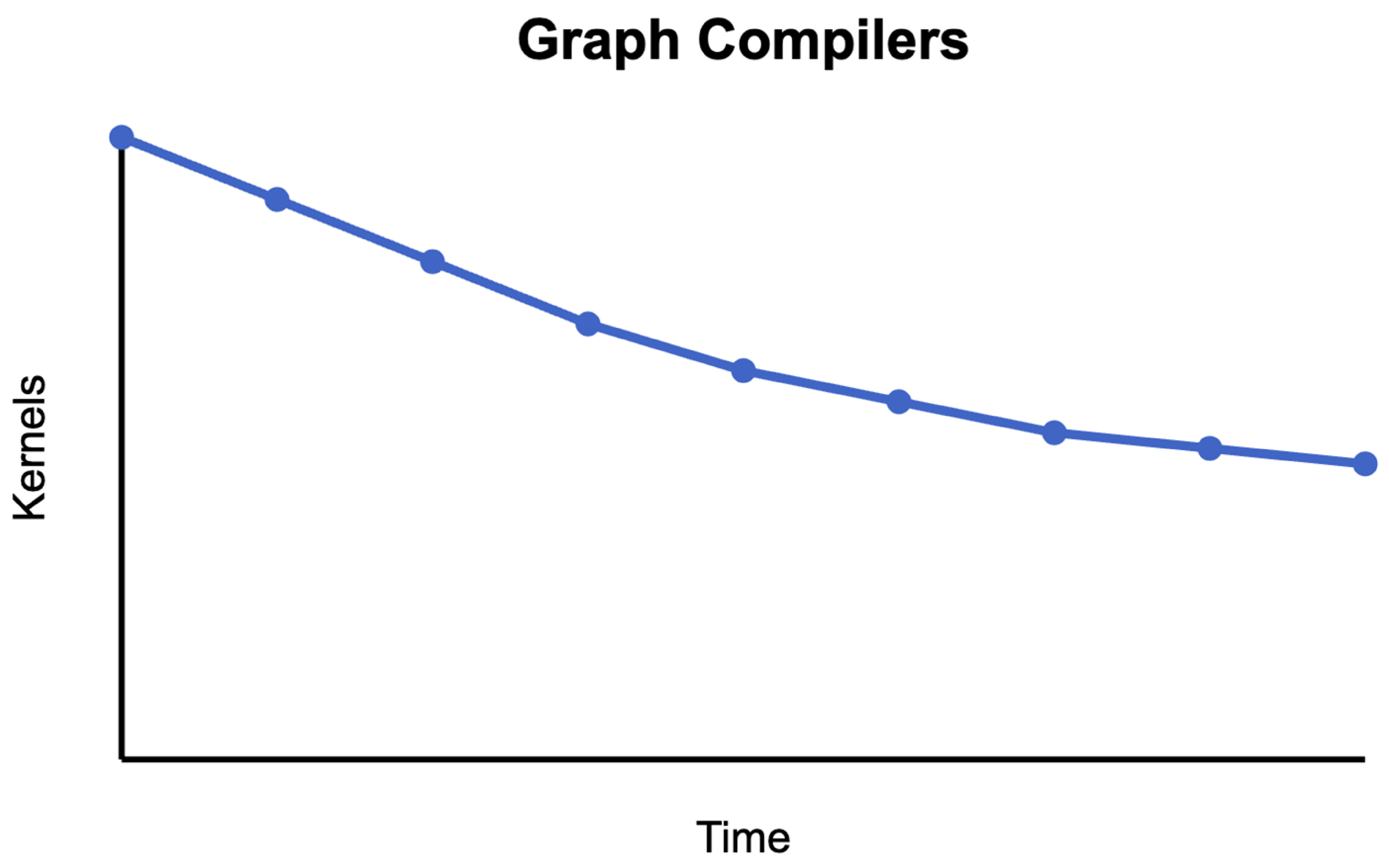
All The Users
Programming Model!
Lower-Level System

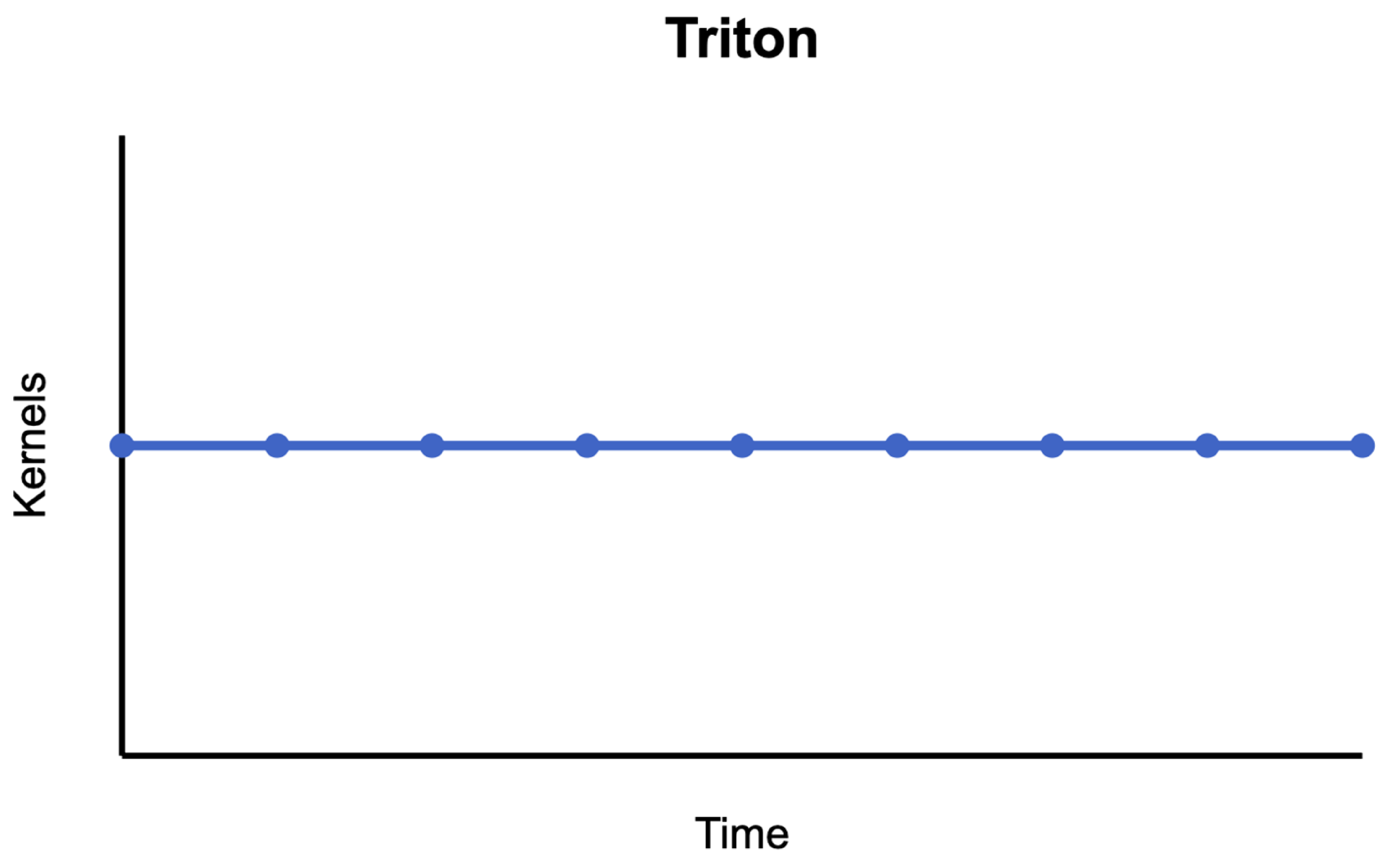
Examples of Programming Models

- Manual Memory Allocation vs Garbage Collection
    - Garbage Collection asks nothing of users, but in exchange you also have no guarantees on when memory will be deallocated.
- Python vs. Compiled Languages
    - Python (more or less) guarantees that your operations will be executed in order, they'll always check their __dict__ object, etc.
- Kernel-Authoring vs. Graph Compilers
    - You need to write a "kernel", but in exchange they often guarantee you 1. one kernel launched, 2. How many HBM accesses are performed, etc.

Providing guarantees is an extremely powerful tool to empower users, but avoiding those guarantees is often how systems can provide value.

Graph Compilers vs. Kernel Authoring





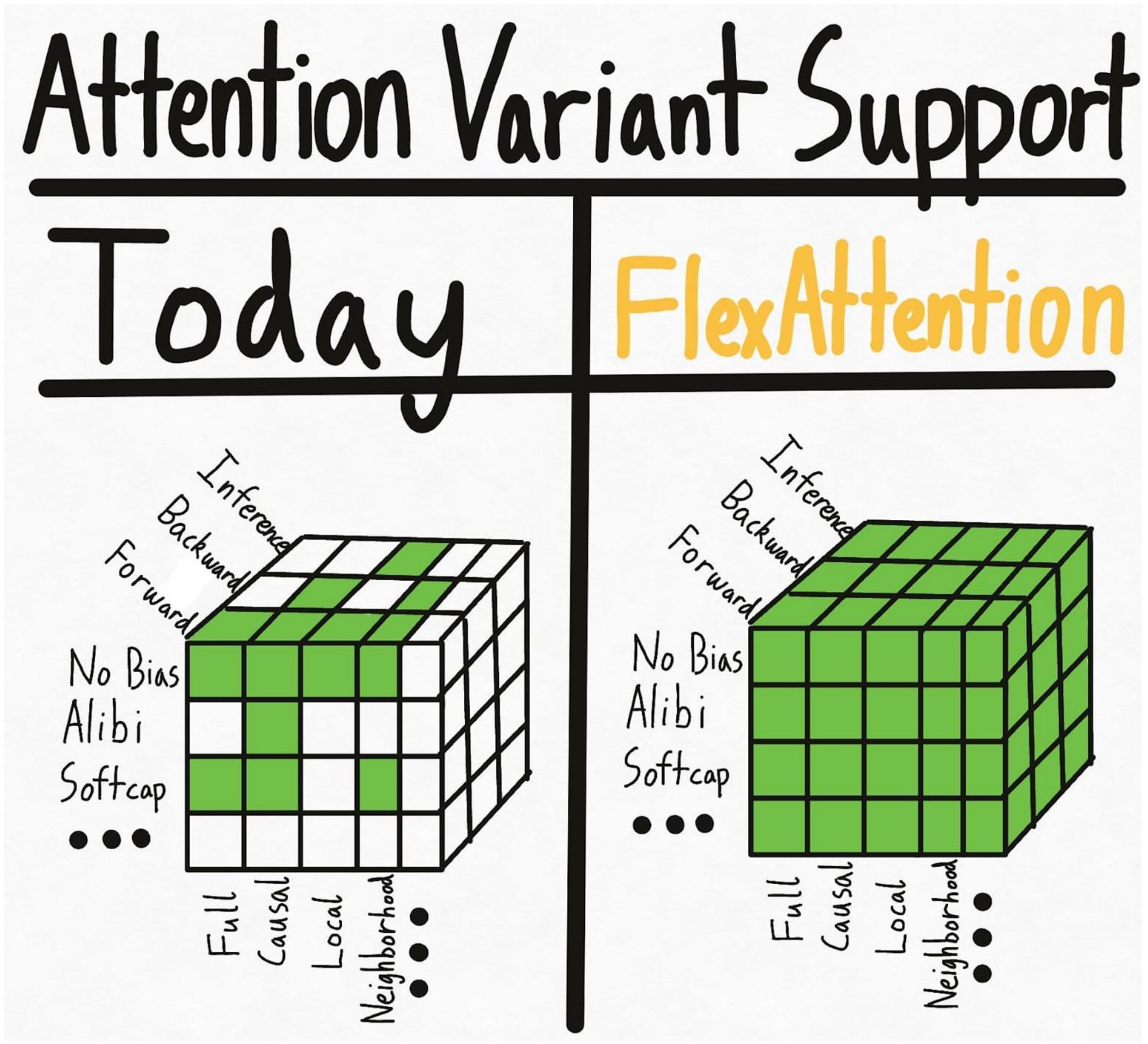
FlexAttention

Guarantees:

1. It’s guaranteed to always results in a fused attention kernel
2. It’s guaranteed to always have the same memory properties as a fused attention kernel.

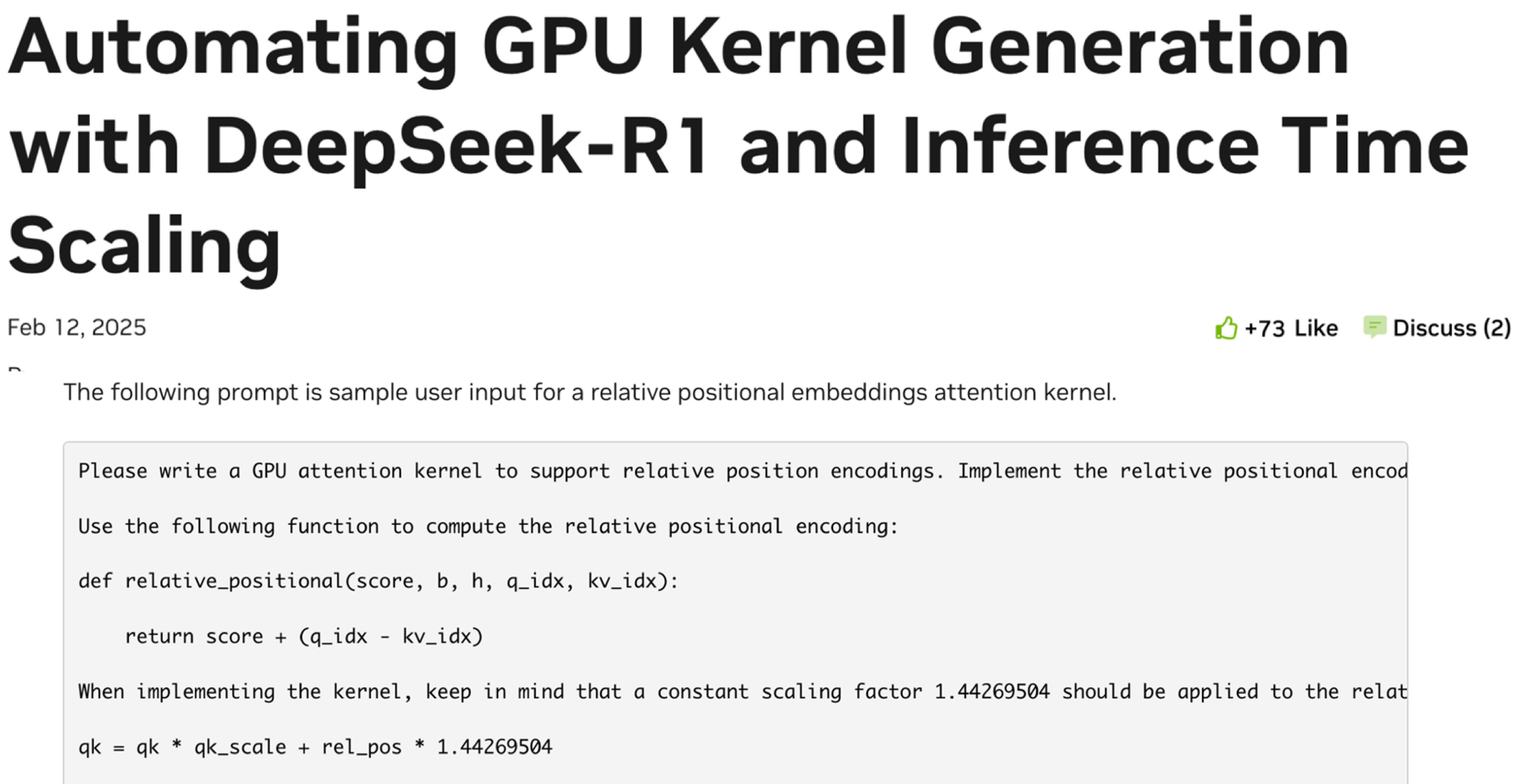
But:

1. We added a CPU implementation
2. We’re adding optimizations to take advantage of warp specialization.
3. We’re working on adding a FA3 backend.



The Right Programming Models >> The Right Implementation

- A programming model is how you separate the concerns of the user from the underlying implementation.
- An AI can write all the underlying code, but for the foreseeable future, it must interop with the user's intent.



Automating GPU Kernel Generation with DeepSeek-R1 and Inference Time Scaling

Feb 12, 2025 +73 Like Discuss (2)

The following prompt is sample user input for a relative positional embeddings attention kernel.

```
Please write a GPU attention kernel to support relative position encodings. Implement the relative positional encod

Use the following function to compute the relative positional encoding:

def relative_positional(score, b, h, q_idx, kv_idx):

    return score + (q_idx - kv_idx)

When implementing the kernel, keep in mind that a constant scaling factor 1.44269504 should be applied to the relat

qk = qk * qk_scale + rel_pos * 1.44269504
```