(no title)

85 MIN · YOUTUBE · HTTPS://WWW.YOUTUBE.COM/WATCH?V=EFM546A79AM&LIST=PLOROMVODV4RMQXOCAZWATUHHQ-YEMBLCV&INDEX
https://www.youtube.com/watch?v=EfM546A79aM&list=PLoROMvodv4rMqXOcazWaTUHhq-yembLCV&index=10

SUMMARY

The lecture focuses on the importance of inference in machine learning, particularly in the context of language models. Inference is the process of generating responses from a trained model based on prompts, and it has become increasingly significant due to the growing use of AI applications such as chatbots and agents. The discussion highlights the challenges of inference, particularly its memory-bound nature, and explores various techniques to optimize efficiency without sacrificing accuracy.

- *Inference is crucial for practical applications of trained models, including chatbots and code completion.*
- *Efficiency in inference is vital as it incurs repeated costs, unlike the one-time expense of training.*
- *Key metrics for measuring inference efficiency include time to first token (TTFT), latency, and throughput.*
- *The auto-regressive nature of inference limits parallelization, making it more challenging than training.*
- *Techniques to improve inference include reducing the size of the key-value (KV) cache, quantization, model pruning, and speculative decoding.*
- *Grouped query attention (GQA) and multi-latent attention (MLA) are methods to reduce the KV cache size while maintaining performance.*
- *Continuous batching and page attention techniques help manage memory more effectively in dynamic workloads.*
- *Future architectures may need to be designed specifically for inference to overcome the limitations of current transformer-based models.*

All right, let's get started. So last time Tatsu talked about scaling laws and we're going to take a little bit break from that um and talk about an inference. Um so the problem of inference is very simple. you've trained a model and you're given a prompt and you want to produce the response usually as accurately and as quickly as you can. So inference turns out to be um only one lecture but it's actually of growing importance and it shows up in many places. So clearly once you've trained a model, you don't just sit there and um if you're a researcher, you put a plot in your paper of here's how my model works. But um for everyone else, you actually want to use this model. So what does use look like?

It could be chatting with an AI assistant or chatbot. It could be doing code completion. Um these days agents are very popular and uh that requires inference. Um batch data processing as well. Um it's also useful for evaluation um for evaluation that requires generation in particular and it's also used inside training if you're doing reinforcement learning you need to have a model you generate rollouts and you score them and you update the weights appropriately. So inference shows up all over the a place and you know efficiency as the is the you know theme of this class really matters more than ever here and even if you just look at the actual use um kind of app uh vantage point you know training is a onetime cost. It could be very expensive. It is very expensive but once you're done with it that's it. But inference is a repeated cost. you you kind of incur them every single day. Um so

you know OpenAI is estimated to produce 8.6 trillion tokens a day. Um and just for reference DeepS v4 which came out earlier this year was trained on 32 trillion tokens.

So in less than four days the number of tokens that OpenAI has to produce and therefore the com compute is um you know at least DeepC V4 and of course you know even the frontier models might be trained on more tokens but still inference um the number of tokens generated is is going to be higher. So that's just a you know some perspective and moreover I think this the importance of inference has grown in the last year because you know once upon a time we thought of language models as primarily as chat bots or assistants where you put in a prompt you get back a response and presumably the goal of the human is you read the response. But now as we move more into a gentic world, what it looks like is that a query goes in and the agent is going to do a bunch of stuff.

It's going to think, it's going to reason, it's going to call some tools, it's going to introspect and at the end of the day, it will produce some output for a human to read. So the most of the tokens that the agent produces is actually not for uh reading. And so you should really think about the number of tokens generated as really the compute spent. And there's no sort of limit. If you have an ambitious enough problem, you're going to need much uh much compute and a lot of tokens, right?

So if you were in the chat world, maybe inference um you after a certain point it's fast enough because humans can only read so fast. But for an agent, you know, there's no kind of limit to how much value you can get out of squeezing more out of inference. So there's a lot of people doing inference. Um on the commercial side, um of course all the the closed API providers have to serve their models. So inference is a big deal for them. And there is also a bunch of providers serving open weight models um and providing inference um as well. And in the open source community, there's a bunch of packages. Uh, so VLM is probably a popular one, kind of a a a go-to. Um, SG lang is another one that's u particularly good for agentic workloads, but maybe not as popular um yet. And then there's tensor RT from Nvidia, which is really, you know, fast, but it's more narrow. Um and then llama CPP if you want to run inference on your um on CPU um this is a popular you know package.

Okay so hopefully I've argued that inference is is huge. Um if you can make inference you know twice as fast or even 10% faster that is a big deal. So now the question is what does fast mean? So there's a few metrics that capture the notion of fast and they're going to be applicable and in different settings and have different trade-offs. So one metric of fast is time to first token or TTFT. This is essentially how long a user waits before any generation happens. So you put in a query into uh TGBT and uh some number of you know milliseconds um um passes by and then and then from the point the first token comes in that is that is the time and this is generally useful for interactive applications because um that latency where you're just waiting and doing nothing is is the longer that is the worse the user experience and as soon as tokens come in it doesn't maybe have to be that fast because if you're going to read it anyway, um you can't read that fast.

So the second metric is latency. Um and this is from the standpoint of an individual user. How fast are tokens appearing for one query? This is also important for interactive you know applications. Um so basically this is how fast the tokens are kind of streaming through. And then a second related concept is is throughput

measuring tokens per second. This is how fast tokens are appearing for many queries. So latency or one over latency and throughput are clearly very related, right? It's it's very kind of aligned of the you know in general many interventions will make latency and throughput you know better. But uh as we'll see later there's actually a trade-off here. And throughput is useful for if you're doing a bunch of batch processing, right? You just want you have, you know, a pabyte of data and you want to process it with a language model. Um, you just want that job to get done. It doesn't matter if one uh query is coming in faster, sooner or later.

Okay. So, what determines the efficiency by either any of these metrics of of inference? Um so the the high level bit is as follows and this is kind of a you know maybe one high level bit to remember from the lecture is that um when you're doing training you see all the tokens at once because in supervised fine-tuning you see all the tokens and you can parallelize over the sequence remember think about in the transformer the calculation for attention MLP um the sequence is just a dimension so it's a big tensor that gets multiplied um And so you essentially process all the tokens at once.

In inference, you can't do that. You have to generate because of the auto reggressive nature of inference token sequentially one at a time. So this is a fundamental problem of why inference is a very different workload than training because you can't paralyze across the sequence dimension. And therefore, as we'll see later, it's going to be harder to have high arithmetic intensity or uh fully utilize the compute. Okay.

All right. So, let's uh let's go on. So, first I'm going to just maybe of a preview of what's in this lecture. Um first, I'm going to do some math to understand um the arithmetic intensity and um throughput and latency. how to think about that for a transformer. Um, and then I'm going to talk about various uh techniques to reduce the uh the cost by you know making the KV cache smaller, quantizing model pruning um and then finally I'll talk about speculative decoding and then some kind of uh practical concerns.

All right, so um this is going to be a bit of a review mostly kind of swapping in make sure we're on the same page with respect to notation. uh much of this lecture is based on the scaling book from Google on um you know transformers and inference. So I do recommend that people go check it out. It's very nicely um written and some of the picture um the figures are um you know uh generously um lifted from that book. Um okay so um just to establish a bit of notation to understand this uh this diagram. Um so we're going to use symbols um think about in ops to denote dimensions um but also their kind of length. So B is going to be the the batch dimension and also the number of sequences. Uh T is going to be the sequence dimension which is also the number of tokens. D is the model dimension. Um H is the head dimension.

Okay. So if I'm going to write this um this is taking a a product between a tensor with dimensions B, T and D. um and another matrix of D and H and I produce BT th. So what is going on here is that there are some dimensions marked red which are going to be contracting. These dimensions appear in both uh operands and disappear from the um the results and then there's going to be other um regular dimensions in black that appear only in one operand and stay in the result. Okay, so this is kind of the general form. Um another case is if you have blue dimensions um that means they're they're batching dimensions and they appear in both but stay in the result they do not get um contracted or reduced. Okay. So the interpretation here as we'll see later is that we

have this tensor for every uh sequence and token position you have a a vector and you want to multiply that by a you know a matrix.

Um and then here you're basically taking a bunch of dotproducts uh between uh two you know tensors. Okay. So with that in mind this in all its full glory is the description of uh a transformer block. Um so it looks a little bit like a circuit diagram. There's a lot of details here but I actually find this probably the most um kind of crisp definition of what a transformer is. you know you see a lot of description of transformer and frankly a lot of them are kind of you know it's hard to understand what is exactly happening this one basically tells you exactly the shapes of the tensors and allows you to reason about the the the dependency so um for a transformer block you take the x um which is the activations of one layer you feed it through attention and then you feed it through mlp um and and just as a kind of a quick refresher um the the x you um projected using a query matrix um a key matrix and a value matrix. Um you'll see here that um we have um n which is the number of heads and h is a dimension of each head.

So the query matrix is um batch time sequence time number of heads times the head dimension. Um and k and v instead of having n we have k which we'll see later. um which we have a potentially smaller number of um key value heads. Um and then this is the tension operation um where notice that there's batching dimensions um so B appears in in both and we're contracting over the head um you know dimension and we'll come back to why this B being in both makes uh inference uh hard um why attention is kind of a bottleneck there. Okay, so just kind of hold on to that thought.

Um and the MLP is is kind of very you know simple and straightforward. Um you just do some you know map malls. This is the um the the gating um um you know matrix. This is the up projection and the down projection. So um I think you guys have implemented this so I don't want to belabor it but just to swap in the notation. Um by convention we're going to assume that um in the MLP um the MLP up projects the D-dimensional model dimension into four times that. So f is always you know think about as 4D whenever you see f um and then the model dimension gets split across the n heads.

So um the model dimension is always equal to number of heads times the head dimension. Um and then the uh the number of uh heads gets split in the case of group query attention into the number of groups and the number of heads uh per uh group. Um and finally we have two variables S and T which both represents the um kind of the sequence dimension and the difference is that S is going to be represent the number of um input tokens to pro and T is going to represent the number of output tokens.

So in training time these two are the same because we're predicting all the out um the same um inputs as outputs. Um at inference time uh t is going to be one and s is going to be the input. Okay, so another review for arithmetic intensity. So I talked about this in the second lecture. So just as a warm-up, suppose I'm multiplying two matrices, a B by D matrix and a D by F matrix. So intuition B is the batch dimension, D is a hidden dimension or the model dimension and F is a projection dimension in the MP. Um so remember what is arithmetic intensity? I have to count flops and I have to count um the amount of memory moved. Okay, so we can we can do this as follows. So um what do you have to do to multiply these matrices?

Um now remember the systems you know lectures um you have to read x from um hpm and if you're storing everything in bf16 which is the case for for inference um we're going to have 2 * b * d you're going to read w that's going to be another 2 * d * f um you're going to do the mold so that's um number of flops is 2 * b * d * f and um and then you're going to write it back.

Okay, so every time you read and write it's basically the basically number of entries. It's a kind of a quadratic like term and mammals are are cubic and remember um this is going to be important because that's how you're going to get high arithmetic intensity. So number of total number of flops is um you know the the cubic and then um the bytes transferred is the sum of the reads and writes. Okay. So remember arithmetic intensity is how much compute we do per bite transferred and we want this to be high.

So the intensity is is going to be equal to um you know flops over bytes transferred. And I'm representing all these things symbolically because it will be uh easier to to see and work with rather than numbers. Um one kind of just you know simplification we can make is that suppose that the batch dimension here is much less than d and f then we can simplify this. So um in particular what this is doing is saying d= c * b and f= c * b and let c go to infinity. So both D and F are really large and B is smaller and this just reduces to intensity of B.

Okay. So the upshot of that is that um remember when we did arithmetic intensity for um um you know matt mole it's was something like n over n over three um and this is the kind of the analog where instead of just having you know one variable for dimin having square matrices we have non-square matrices. Okay. So just to compare so the accelerator intensity of uh um hardware is um you look at the flops per second in in the in the spec sheet. You look at how much memory bandwidth how fast memory gets moved between HPM. You divide and that is the accelerated intensity.

And now you compare your computational computations intensity with your accelerator intensity. And if it's greater then you're compute bound which is good. If you're less then you're memory bound and that's bad. So in this case uh for H100 for this particular map operation you're compute bound if um the batch size is greater than 295. So here's an extreme case. So suppose you just have one example. Okay. Um what happens? you have one example, then your arithmetic intensity is going to be one and this is going to be memory bound.

Um, and you can kind of see what's going on here. You're reading this DF matrix. Um, but B is only one. So you're effectively doing only two * D * F flops. Um, so you know that's and this is basically kind of the workload that you'll see in inference. You don't get these like full matrices. you get these like very thin um matrices or tensors. Okay, let me stop there in case there's any questions. Yeah, in the transformer I just want to confirm sorry uh across across.

Yes. Can I search cross? Across across across across you can't just cross. [clears throat] Yeah. Should that be across K moves instead of G? I'm having trouble with the connection. [snorts] >> Um, let's see. So, the number of groups here is uh wait, I'm just making sure. So, K is the number of key value heads and um so you have Q sorry G is that G per KV head. Um so >> my understanding is like each key had >> yeah correspond if we break down the n into a g then that means g is the query i that means we have ks that's my >> okay so um I think yeah

I think you're right so k should be the number of uh groups and g should be the number of uh the you know heads per one of those groups.

>> Yeah. Okay. Thanks for that. I'll fix that later. Okay. Um so let's talk a little bit more about the arithmetic intensity of inference. Okay. So um here's how uh to think about what's happening in inference. Let's say you just do it naively. So here is um a prompt never going to give you um going to the transformer um and the transformer is u does you know generates keys and values and activations and at the end of the day it generates uh logits over the output vocabulary and you sample a token and that token um gets concatenated with um the prompt and then you just do this again. you generate never, you attach that to the prompt and so on and so forth. Okay, so this is the most naive thing. If you have a black box that takes a sequence and outputs a distribution of a tokens, which is exactly what transform does, you can just apply this repeatedly.

Okay, so this is uh works but this is really bad because each time you generate one token um you actually take order t^2 time where t is the number of you know tokens that you've uh generated so far. So generating t tokens is actually key uh cubed. Okay, because the the attention is already t squared and you have to do that you know one for every once for every token.

So this is um you know pretty bad but the observation is that you don't actually have to do this. Um a lot of the work can actually be shared across prefixes. So for example, if you're over generating never versus you're generating up, you're actually computing a lot of the same key values for um you know never going to um give. So these tokens they shouldn't change. Okay. And this is because it's a causal you know transformer. If it's birectional then if you attach a token then everything changes. But if it's causal then the activations here don't change um based on any tokens you append.

So with that observation, the first obvious thing is to store a KV cache in HPM so that between successive token generations, you can just reuse the KV cache. So that means when you are um trying to uh you know generate GANA, you don't actually have to compute all the keys and activations of these previous um previous uh tokens. So this is what it looks like with a KV cache. Okay. So um there's going to be two stages prefill. So you get the prompt and you populate the KV cache which is the set of key and value pairs that the transformer computes. Um and then you generate the logic just the same computations we did before. And now um now you have the KV cache here um and then this distribution which you sample a token. And now you feed that through the transformer which you know uses this cache um and then produces a both uh the distribution of the next tokens but a new KV which is the the basically activations corresponding to the token up and then this gets fed in. So now you have the commmented KV cache, you have this um logits, you sample and then that feeds to the transformer. You output the distribution of next tokens as well as the the next um the new token which you add to the KV cache and so on so forth.

Okay. So the KV cache formally is for every you know sequence so there's B of them for every token um there's s of them and for um every layer in every head you store each dimensional vector. Okay. And just to reiterate for inference, there's prefill. You're given a prompt. You encode um the prompt into these uh you know vectors. And this is parallelizable just like in training because you see the entire prompt, you can compute the um the KV

cache and then in generation you generate the new response tokens, you know, sequentially.

But at least you don't have to pay for um generating the KV cache of the tokens that you already looked at. Okay. So, is everyone comfortable with a KV cache? Any questions about this? All right. So, let's try to figure out um the flops and memory IO for both MLP and attention layers. Um, so remember S is the number of tokens we're kind of conditioning on and T is the number of tokens we're generating. Um, so we're going to do this sort of abstractly and then later we'll specify to the prefill where T equals S and generation T equals 1.

So for the MLP layers and through all of this I'm just going to look at the mammals because the the mammals are the thing that actually uh require a lot of work. everything else can be um is not that many flops um and can be potentially fused into the mammal too. Um okay, so let's do this uh calculation. So it's the same as the matrix uh calculations. I'm not going to maybe be labor step through every single detail here, but just the form is you read X from um high memory. um you read all the parameters, you you compute the um up projection um and then you know you write it to HPM, you compute the the um the gate um and you write it HPM and then you uh compute the basically the down projection of that and you write it um to HPM.

Okay, so the number of flops is uh you know depends on P B and T and DNF. So batch size, sequence length, um model dimension and the fe forward dimension uh MLP dimension and the bytes transferred is um you know this expression. Okay. So now you can compute the arithmetic intensity which again is flops divided bytes you know transferred. Um so this is some expression. Um we're going to assume that just like before B * T is much smaller than DN DNF. Um and with that then we see that intensity is um you know B * T.

Okay. So this is analogous to just the ML case. There's more matrices and there's like more dimensions but fundamentally it's kind of the same thing and it makes sense. MLP is basically a big mammal. Okay. And it also makes sense because the the batch dimension and the sequence to length dimension are sort of everything is independent right in for the MLP um they don't interact. Attention's different story and so you know B * T so as long as you have um you know in the prefill if you make B * T large enough large batches long sequences you'll be fine. Now let's look at generation. So remember generation um there's two problems here. One is that gen generation t equals one. Okay, you're only generating one um token at a time. And uh so that means your arithmetic intensity is going to be B.

But B what in generation is the number of concurrent requests. And so if you're in a sort of batch setting, you kind of can control that. But if you're let's say serving a chatbot, it's the number of concurrent requests is essentially um you know how many users concurrent users there are which can be high can be low. So it's a bit unpredictable it can be changing over time. So that's something we're going to address when we talk about um um you know continuous batching.

But overall, this is not bad, right? Because as long as there you have large batches, um the sequence length isn't going to really help you, but you have large batches, you should be good. So now let's look at the attention layer. Um so again, S is in the previous tokens already generated. T is the number of tokens you want to generate logits for. Um so in attention you read um the the QV uh you know matrices from HPM um you compute the

tension um you you know compute the softmax which doesn't really matter um and uh compute the the value you know matrix and then you write the result to HPM okay so the flops is B * S* T * and the bytes transferred is uh this expression.

So this should you know all the the everything is a mammal right so this is should be um always you know one degree higher of a polinomial than um the bytes transferred because it's a mammal. The only question is you know what is that factor look like? So that factor for attention is s * t over s plus t. Okay. So, so let's look at the prefill. So, prefill um what this is saying is that the pre-fill intensity is um is s over two when t equals s. So, this is good, right? Because as long as you have long sequence lengths um for attention, you're going to maintain high arithmetic intensity.

Um notice that the batching dimension doesn't um happen here. I'll explain a bit you know why um later but for generation this is bad news right so the the generatedation intensity is s over s + one and that's less than one or even let's just call it one and arithmetic intensity remember one is bad we want it to be something like 295 for h 100 to be to saturate the compute and so you know this is really the bottleneck.

So we did all this analysis um for you know MLP attention um you know pre-fill generation and we've found this to be a bottleneck. So let's try to contemplate why uh this is a bottleneck. Um so unlike MLPS so MLPS for generation is actually okay as long as your sequence is um sorry is your um uh you know batches large enough. So the problem with um this uh ML you know attention is that you know let's look at MLPS every sequence hits the same MLP weights.

Okay, so these don't depend on B. Um whereas in attention layer each sequence has its own KV cache. So these all depend on you know P. Um and so you can think about this as um what's what's going on here is that you know in the MLP case having B being big actually is is helpful because you kind of get to load these MLP weights you know kind of once um and you can kind of use it for all your sequences right that's how you get high arithmetic intensity because you use a simplifying a bit you load it once you do all your batch processing and then that that is that is good whereas in attention you can't you know all these depend on B so increasing B doesn't help it's like every for every sequence you're basically doing a matall so they're all in independent so you know doing more matt malls isn't uh helpful just like remember in the very beginning um I showed you this example, this also has pretty bad uh arithmetic intensity. It's not not a mammal because we're essentially batching by um a coordinate.

And this is essentially basically the same as doing a dot product, which has horrible um uh arithmetic intensity. And remember this is in the attention. This B blue B is like the cause of why um the tension is uh arithmetic intensity doesn't scale with B and why that is a bottleneck. Okay to just summarize here so prefill is compute bound generation is man memory bound. So if you look at the MLP intensity for the um for prefill it's B * S great prefill for in um for attention intensity S over2 not as good but workable um generation MLP intensity also workable requires long uh concurrent requests but it is really the generation intens attention intensity which is a fundamental bottleneck and you just that's it. You you just have to if you're sticking with a transformer, you can't uh really, you know, improve this.

Okay. Uh pause there for questions. So now whenever you hear people say, oh, inference is memory bound, um you know why. Okay. So let's now use these um intuitions and calculations to think about our um you know inference metrics throughput and latency and also TTFT.

All right. So inference is memory bound. um the the now the main I mean in some ways this simplifies a lot of things because when we think about how long things take you just look at how much memory needs to be transferred okay because uh assuming you overlap uh communication and computation the bottleneck is going to be just the amount of memory that you have to deal with. Okay. So in some ways it's nice because it's simple but in other ways it's um frustrating that your accelerators are sitting there not doing anything.

Um okay so let's walk through example. So for llama 2 13B um on a H100. So what is the latency and the throughput? Okay so remember llama 13b um has a particular shape. So oops um this is a sequence length the model dimension fe forward dimension number of query number of key value heads there's no GQA here so the nn equals k head dimension number of layers vocap size um and the memory bandwidth for h100 okay so um so now using this uh this config let's compute the to transformer performance um you know statistics okay so these are are inputs here um and just what the statistics I'm going to compute are going to be um number of parameters um the memory usage latency and throughput okay so first of all um you know what's what uh does takes memory Okay, so the parameters take memory. So we compute the number of of parameters. Um and so you have look at the the embeddings the um the you know the MLP layers that um the the projections of the KQV um at the end of the day you get um some number of parameters.

um assuming that uh you know we're doing a BF-16 which always case for inference then parameters take up this many bytes also in memory is the KV cache um and the KV cache is uh the number of tokens in your sequence times um the number of um your heads and the KV heads um times the head dimension times the number of layers ers you have one for the key and one for the value and then you have two multiple two for BF-16 okay so that's the size of the KV cache and the total memory usage is uh this is for each sequence you have B sequences so that's B times that plus the parameter size and that's the amount of memory you you need um and now what's the latency the latency is determined by uh the memory a IO. Okay. Um because inference is memory bound um most assuming you overlap communication and compute um all the memory is it's it's going to be on um you know shuffling parameters um you know back and forth between HPM and um SRAMM.

So um that's how long it takes to you know move memory around. Throughput is the inverse of latency but we're generating B tokens in parallel. So this is um you know this is tokens per second. This is seconds per token. This is tokens per second um as well as having an additional B because you're processing a batch of B. Okay. So now let's um compute for this config what are these actual values.

Okay. So num parameters is um you know 13 billion. So good sanity check that is a advertised as a 13 billion parameter model. Um memory is um this this term which is about 838 I guess million times B. So it's basically you know a linear function of B plus uh some other term. So this is a KV cache which grows as B. This is the the parameters which is double the number of params.

Now latency is uh just this scaled by the memory bandwidth. So it has the same form as as the memory. Um and throughput is B over that. Okay. So notice that latency as you increase B grows um because the KV cache grows and in order to process stuff you have to copy the KV cache back and forth. And now uh throughput is more interesting because as you increase B you know throughput does improve um but up to a limit right because throughput in um improves because now you're advertising the cost over a larger batch um but also you know the speed at which you're processing this obviously is also um increasing over over time. So um this sort of asmmptotes it's not going to you know through can't possibly go to infinity.

Okay any questions about uh this so far? So basically you just have to remember latency is a linear function in B. This is the where the constant is the size of KV cache and then this is the number of parameters and then throughput is um you know some B proportional to B over B plus something. Okay. So now let's instantiate this for a bunch of situations. Okay. So if you have batch size one um so this is what you would get. uh you get a latency of 0.08 um you know seconds per per token um and the throughput is 124 tokens per second.

So if you what happens if you increase the batch size you'll see that um the latency goes up um but the throughput also goes up. So latency gets worse but the throughput improves. So this is kind of an interesting thing because we think about oh we just want to make it go fast but fast actually has two meanings here which actually depending on which one you care about is you know complete opposite. If you want to tune your batch size um that's going to really determine um you if you want latency or throughput and what happens if you increase the let's increase we're in a we really want high throughput because we're processing lots of documents. Let's increase the batch size even more. Okay. So the latency gets even worse. Not not I mean yeah it gets worse. Throughput gets even better. But the main problem here is that your uh memory you run out of memory. Okay. Because the memory uh for storing this KV cache is ex going to exceed your H100 memory.

And if you have B200s you can increase the batch size more but eventually you hit some sort of limit. So there's kind of limitations to how much you can in through your throughput. You'll never get to sort of the asmtote um because you'll hit uh the memory. Okay. And also your your throughput you know is getting your gains are kind of diminishing as well. So increasing batch size worsens the latency because now you have a larger KV cache to read and write. Um and remember it's batched. So if you're an individual query you have to wait for everyone to finish. So you have to basically you get you're like waiting for a bus and the latency is you know pretty high. You wait and then but uh and then you go whereas the throughput of a bus is pretty good because you can move everyone at at once. Um so uh the the throughput improves as batch size increases because remember the parameters uh are are shared and you load that you know once um into into memory and you can process a lot of um sequences.

Okay so trade-off between latency and throughput just to make sure everyone's aligned. Smaller batch sizes use better latency but worse throughput. Large batch sizes yield better throughput but worse latency. Okay. Um I'm not going to really talk about parallelism too much. There's also another dimension of inference which is um you can you know shard your your model across multiple um you know devices. Um you can look at the scaling uh book chapter on inference if you want to know more. Um just as a kind of a very trivial example, if you launch five m copies of the model, the latency is the same and the throughput increases by by m. Um and then the other

metric we didn't talk about is time to first token. And this is essentially the time it takes to do prefill because after you finish prefill then you can basically start generating.

Okay. So um if you want uh you know to fast to um if you want you know faster TTFT you should use smaller batch sizes um and you want larger batch sizes to uh improve throughput. Okay so hopefully that is uh clear. Any questions about throughput and latency and how they're in tension with each other?

Okay. So now now we have a conceptual framework for thinking about how efficient inference is in terms of arithmetic intensity um and you know throughput and and latency. Now let's try to make it faster. How do we make inference faster? There's a bunch of different techniques which are quite varied ranging from changing the model architecture to doing systems optimization everything in between. So inference in some sense is a fairly rich crosscutting topic.

Okay. So the the most kind of d maybe now in hindsight obvious thing you can think about is hopefully beat it into your head that um you know memory is the bottleneck for inference and the KV cache takes up a lot of memory and it could even be larger than the number of you know parameters if for a large enough batch size. So let's just try to reduce the size of a KV cache. Now you have to be careful about how you do this because you want to make sure you don't lose too much accuracy in the process.

So here is one thing you can do um which we already talked about which is grouped query attention um and just as a kind of a reminder here. So multi-headed attention um basically for every token you have a key and a value and a query. And if you do grouped query attention, you basically compute um the same number of queries, but you have only a smaller number of groups and you compute key and value for each group.

Um okay. So so K is a number of of groups here. Um and so in the multi-headed attention K is N. No reduction. Um there's something called multi-query attention which no one uses because it's really bad. K equals 1 and somewhere in between is hopefully where we'll find a balance between accuracy and speed. So there's this this is the paper that introduces GQA from 2023 and uh they show that if you look at you know time uh per sample um which is related to both latency and throughput um you know you see that um the the MHA so multi-head attention full attention is has this high uh time whereas um if you start k equals 1 it's it's much faster And then you can actually keep on increasing the K to like you know K equals 8 and it's still pretty good and eventually um your your uh time goes up quite a bit.

Okay. So um you know why does GQA improve latency and throughput? Well, it reduces the KV cache by a factor of N over uh K. Um and you know and re just as a friendly reminder re reducing memory usage leads to speed up because we're memory bound. Okay. So let's revisit our friendly llama model here. Um remember in the initial configuration we're just using k equals uh n. So there's no multi-headed there's multi-headed attention. No um no reduction in number of uh key values. Um and so for this one um remember the using a batch size of uh 64 we get this throughput and latency. Now if you do GQA we're going to put in let's say um um a kind of a sparity of one to five.

um that reduces the um you know the memory quite a bit which in turn improves the latency and also improves the throughput. So it's not that latency and throughput are always at odds. If you reduce the amount of memory then it improves both. It's mainly the batch dimension that allows uh that is the point of tension. Okay. So this is this is great. Um and let's actually just imp increase the batch uh size even more and see what happens.

So um before if we had a batch size of 256 we ran out of memory but now it fits in memory and uh we we see that um you know the latency you know suffers a bit because we're increasing the batch size but the throughput um you know goes up proportionally. Okay. So sometimes you kind of play with these uh parameters jointly like you can reduce the KV cache but that allows you to increase the batch size and allow make making other trade-offs.

So the final thing you have to do whenever you do some you know lossy change is that you make sure the accuracy doesn't drop and this paper shows that for GQ GQA um the you know the time is is better but the the um the across a bunch of evalu basically it works well. So now with these accuracy evalu I think you always have to take it with a grain of salt because this is like for particular model later the deepseek paper was to show that it actually does hurt. So you know I guess take everything that's not just kind of math with a grain of salt here.

Okay so speaking of deepsek um here's another idea to reduce the KV uh cache. Okay, so the theme is reduce the KV cache and latency and throughput improved. Um so this is um multi-headed attention same number of queries and keys and values for every um token and um GQA remember we have reduced the number of keys and values for every token um sorry not for every token we basically have reduced the number of values and uh keys and now the multi- latent attention for deepseek says we're actually going to leave the uh number of keys and values um this the same one for every you know token essentially but I'm going to parameter I'm going to compress these so normally um you have some um how do you compute your keys and values you have your um your activations and you multiply them by some matrix to get k and some other matrix to get v and these are generally n* h you know dimensions like your model dim which is which is big. So MLA says I'm going to actually project this um these activations down into C dimensions. So deepse V2 reduced it from you know 16,000 to 512. So this is quite aggressive compression here. And then I'm going to compute the K and the V from this compressed representation.

So now I can just store C. This is much smaller. And then when I need my keys and values, I can just materialize them. So there is one wrinkle here which is that MLA is not compatible with rope which operates directly on the keys and values. Um so what they do is add additional dimensions um for you know for for handling the rope. Um but more or less it's it's still a pretty big you know reduction. So the latency and throughput improvements um follow by just you know simple math. The smaller the KV cache the you know the faster you go okay it's almost kind of uh linear scaling up until some point and then remember you need to check whether your model is um you know accurate. So um first of all this is sort of the result that you know kind of contradicts the or or is intention with uh the GQA paper.

um they show that GQA actually isn't that um you know great um it you know it's um you know it's much so this is MHA this is GKA these numbers are smaller than these numbers um but uh they show that their method

MLA multi- latent attention um works um even a little bit better than MHA but let's just say it's about the about the the same so This column and this column are much better than I guess there's no GQA on this table. It's uh yeah compare over here.

Okay. So I'll show you an yeah >> how does this compare uh to reducing the number of dimensions the model? >> So question is how does this compare with reducing the dimension of the model? Um so that's a good question. And I don't these oblations don't show that. U my guess is that reducing the model dimension just makes things you know worse if you because you're sort of like indiscriminately just like reducing everything. Um I think the the trick in all of this kind of is to find places of the model where you can squeeze. Um and this aperary I don't think you can necessarily know for sure. You just have to do a bunch of experimentation and see what works. So here's another idea to reduce your uh KV cache. This is called cross layer attention. Um so the idea is that normally every layer has KV uh K's and V's. But you know, let's say instead of doing that, we're just going to compute um KVS for a subset of the layers and then just for the this this layer, I'm just going to use the previous layers um KV cache.

Okay, so um so this is kind of another way of sharing. Just like GQA shares KVs across heads, now I'm sharing KVS across layers. Um and empirically this paper shows that um you know doing this improves a paro you know frontier. So um each of these these models is better than um you know in given a method you can always uh kind of sweep this the size of the KV uh cache by changing the the K and um the head dimension. um which I guess kind of relates to your point about changing model dimension um and then but if you do this uh uh CLA cross layer attention it's uh it's better okay so let me give you uh moving on um whirlwind tour of different techniques for reducing KV cache there's uh local or sliding window attention this is a fairly kind of old idea and a very um kind of natural idea Okay, so um if you look at the full attention matrix is n squ and instead of doing that if you're going to generate a token you just look at the last k tokens. Okay, so you essentially have a sliding window for every token you generate. You just depend on the last um k. Um and now if you do that the the effective uh so the kvcast is now independent of the sequence length. It's just you know um the number of uh um the batch times the the other variables um which is great and this is especially great for long context. Um now you because of number of layers the effective context length as actually larger than the number of the stated context length because information can propagate um you know farther in if you go down the layers. Um now you can do fancier things like you can maybe not do a dense uh selection of layers but you can space it out. You can also do this global plus sliding window where you [snorts] have attention to a fixed um you know grid of different you know token points plus a local uh sliding window. So you can do various things.

Now the problem with this is that it actually still hurts accuracy. So this reduces expressivity. there's no you know free lunch here. Um or at least this was an expensive lunch. Um so the solution that people come up with is that they interle local attention with uh global attention. So these hybrid models have uh you know full attention for some of the layers and some local layers for you know some of the other layers and you're basically always trying to and then so that allows you to essentially um you know reduce the the KV cache a little bit and you're always trying to balance uh you know accuracy with speed.

>> Yeah. the trade-off between like a linear attention variant versus like a sliding window. >> Yeah. So question

is what about um linear attention versus sliding window attention? So um I'm not going to talk about linear attention but very quickly um there's a bunch of uh methods that where instead of you know storing a KV cache is you basically um compute some sort of like compressed representation of all the history. So linear the most naive linear attention is uh you just sum the KV uh values up into a single vector. So that's definitely independent of the sequence length. You can do fancier things. There's like gayet, delta nets and mamba which allows you to um try to compress but not forget you know as much. Um now the question is you know how do those compare? Those have also been used in in place of sliding window um attention um and people have gotten good results with them. You can also use a combination of full attention, sliding window intention, and the linear attention because they sort of capture different, you know, aspects.

If you care about kind of local kind of high resolution stuff, then sliding intentions better. If you just kind of want like broad summaries of the past, then um the other, you know, linear attention might be better. So then for like a long context sentence would you say linear attention would be a better setting for that? >> Um so the question is for long context would linear attention be um better. Um let me talk. Um so there's no free lunch, right? Like I think let's say you have a very long context and you're sort of solving a needle in a hay stack problem, right? If you have have to compress your entire history into like a small context, you're just going to lose information and um you might just, >> you know, not be able to retrieve it.

because I guess I feel like what it seems like people are going with hybrid art tickers that you'll always need some sort of longer attention. But I'm I'm just trying to understand like in my head like what the trade-off between using a sliding window versus like some sort of mamba delta net layer would be that like is using a delta layer like better than using sliding window consistently or like what is the actual like maybe representative tradeoffs?

Yeah, I I guess maybe I'll say that the mamba and delta net are more powerful um than the sliding window attention. Maybe you can think about the mamba as um it'll probably can like it certainly can represent some of the aspects of sliding window attention because like you can just as you're doing the recurrence it can just look at the last you know state. So um yeah maybe yeah maybe you can say think about the the linear attention or it's like extensions as being you know you know better at least there they have more room like once you do sliding into attention like you're done there's nothing else you can >> yeah okay so let me just quickly go through this uh just highlight this you know deepseek deepseek continues to um you know innovate different types of attention mechanism S um so remember they came up with the multilaten attention which compresses the key values. Um there is this thing called um now they have compressed sparse attention deepseek sparse attention and heavily compressed attention. I never kind of remember all these acronyms and what they um they mean but let's look at this diagram. So normally you have your uh KV tokens um and your your query token. So um compressed attention is going to basically compress every M tokens into you know one token um and then there is this thing called deepse sparse attention which um basically selects a subset of those um to kind of keep and the way you select a subset is that you actually compute some light lighter weight queries and keys and then you do like a you know a smaller attention to get these index scores. so you know what to keep. So like a kind of a lightning fast way to figure out what tokens you need to keep and then you use those. Um and then there's a more compression that you know happens.

Um so in the interest of time I'll move on. Um the goal of this section is to reduce the KV cache. Um because KV cache is related to memory and we saw that inference is memory bound. So um that directly translates to improvements in throughput and uh latency and the key is to do this without hurting accuracy. So you can do lower dimensional KV uh caches across layers um by across uh heads um and across like you know head dimension. You can do local attention, you can do linear attention which was discussed earlier. Um and there's much more and also there's also diffusion models which um is a non auto reggressive way to generate which can be much um faster. Okay. So let's talk about a few other ideas which are important. So quantization is more of a I would say less of an architectural and much more of a systems u uh perspective on how to make things you know smaller.

So um the key idea here is just reduce the precision of numbers. Less memory means higher latency and throughput um and the obviously you have to worry about accuracy. So quantization um um is um you know there's many options here all the way from PF16 all the way down to in um in four. Um and so one thing you can do um if you're kind of scared that quantization is going to mess you up is that you train a model with quantization in mind. So this is called quantization aware training. Um and during the fe forward pass during training you quantize and dequantize and you're basically simulating these quantization errors as you train. Um so then you generally now the weights are adapted towards the quantization things will work better but the con is that requires expensive large scale training and typically what so what typically people do is they they train models and then they um quantize after the fact. So this is post-training quantization. It's much uh you know cheaper often. Um you know there's a naive way to do it is that you basically uh for every layer or tensor you basically determine the scale and the zero point for each let's say tensor and then you you quantize um that you know separately. Um this is generally you know doesn't work as well. Um you can use um this idea called GPT Q which uses some hashing information to you know quantize layer by layer and then the you keep track of the errors which get um propagated into the non-quantized weights and so it kind of allows you to correct for the errors. And now activation aware quantizing is um um you know more s kind of a sophisticated um way where the observation is that some activation channels are large and those the weights that interact with those matter more. So let's allocate more precision to these weights. So let's look at this picture. Um so normally if you take FP16 um weight matrix and you quantize it you get let's say you quantizing down to N3 but um what you're going to do is you figure out which of these um you know activ so these are maybe activation channels and some of these activation channels are large so as if they're large in general then you basically allocate like let's say FP16 for this channel you keep everything else as in three and for a few important channels you um you um use higher precision.

Okay. So another idea here is to do model pruning. So here you just take a large model and you rip out pieces of it and you fix it up. It's kind of um a very crude way but it it turns out to work. So there's this paper from um Nvidia where um essentially let me actually um uh let's see so you kind of have first have to estimate the importance of the different parts of the model um and you know choose the most important parts and then you basically remove um different you know um hidden units and different um even uh you know layers um And now now we have a model. It's not going to be very good. Um and so what you do is you post train it. You train it some more on the data or the tasks that you care about to kind of heal it. Okay. So this is in some sense a training uh p way to um to you know reduce the KV cache but where you sort of initialize it with parts of a of a good model.

Okay. So this seems to work pretty well. So they were able to take a 15B model and reduce it to a 8B model um and and it doesn't really hurt accuracy um by too much and the amount that you use to kind of train the model or through this process is um much less. Okay. So just to summarize here um you know the game is to reduce the uh inference complexity without hurting accuracy. You can think about this is mostly reducing either number of parameters or KV cache. Um you can define a faster model architecture and train it or you can um define a faster model architecture, initialize the weights from original model which might have a different architecture but you kind of just make this Frankenstein thing and then you repair the faster model with um distillation.

Okay. Yeah. >> Can you say more about how you distinguish the important layers from the unimportant layers? Um so how do you distinguish the important layers from the unimportant layers? So um you know in general you have you know a calibration you know set and you pass the the the inputs through the model and you're basically looking at you know the magnitude of the activations and the ones that are you know some of them especially if they're dead units will be kind of close to zero and the ones that are large you want to keep. So that's a high level idea >> I guess like why does it matter if the activation is like high it could just be like like what if it's just always high for instance >> like is that >> so in g this okay so the question is why what if all the activations are high so in general this is a kind of an empirical observation that um some of the channels will be much higher than others if this weren't true then these techniques wouldn't necessarily work um but it happens to be true because um you know that's how these models ended up being you know trained um and then you can exploit that >> or I just made up like say a neuron which is always like value 100 for like across all the samples >> would that mean that it's necessarily meaningful or like maybe it's just an artifact of training just like >> I see so the the question what if a neuron is always 100 um I mean if that's the case you can look you can also look at kind of variance um related questions. Like if it's 100, you can't just like remove it because then everything is going to be broken. But you can if it's high mean and low variance, maybe there's another, you know, a way to just incorporate the bias essentially.

Okay, let me quickly go through this other idea. So, so far we've looked at lossy methods which basically kind of really crunch down the KV cache, but it could hurt accuracy. There's a very elegant way of doing this in a lossless way. This is called speculative sampling or speculative decoding. So um remember that if you're doing prefill you can code all the tokens in parallel and this also gives you probabilities. This is fast. This is computebound and all nice things. And in generation it's one at a time. Okay. So checking is faster than generation. If I give you a sequence it's fast to tell me how good it is.

much faster than it is to generate one at a time. Okay. So, you can exploit this asymmetry using the following idea. So, what we're going to do is use a a cheap draft model to basically generate from the guest a few tokens. Let's say four tokens and then we're going to use the model we actually care about the target model Q which to basically review these tokens and accept or not accept them. Okay. So, so things are kind of chosen to be balanced, the draft model is smaller and cheaper. And so even though it's memory bound and it has to generate one at a time, it's not too bad. Whereas the target model is big and expensive, but what we're doing is asking it to process a batch of tokens in parallel. So it won't also be uh too bad. Okay. So here's the the video that shows kind of how things work. So if you use a big model token by token that's it's going to be pretty slow. But if you are um doing speculative decoding then um then you can see the small model kind of uh generating a bunch of tokens um and then the large model uh basically you know critiquing them and then so you basically can get this

sort of burst of tokens and then maybe another burst of tokens and so on and so forth.

Okay. So, um here is the the algorithm for there's a few papers that came out with uh specular decoding around the the same time. This is, you know, one of them. Um and so the idea here is that um in order to generate we're going to generate K tokens um and we're just going to sample from this draft model. So Q is a draft sorry P is a draft model. We're going to sample K tokens and then in parallel we're going to um compute the logits of these draft tokens using Q.

And then now we have to determine whether we accept or not. And this is where you do a bit of math and you know probability and statistics. You basically are going to accept with probability min over this ratio of Q over P. So if Q is um you know much larger than P the larger the Q is the more likely we want to accept it um uh and and otherwise um you kind of sample from this you know residual distribution and exit. Okay. So this is basically rejection you know sampling um except for in rejection sampling sometimes you you you just when you reject you get nothing but here we always uh are guaranteed to get an exact sample from the target model. Um I'm going to skip this um you know kind of simple proof. Um you can it's basically the same arguments as kind of rejection sampling to show that it's the exact uh um you know probabilities from the target model. Um and the initial paper shows that this is um this fast and generally if you look at there's a you know if you have too few draft tokens you're not really leveraging um the batching on the the target model side and if you have too many then you're going to reject more often. So there's a sweet spot around you know in this case three or four.

Okay. And in general the draft model is much smaller than the target model. And ideally you want the model draft model to be as close to a target which means that um you want to kind of distill it. So which means that actually a lot of the same ideas that we just talked about are applicable to speculative decoding as well. So basically the the idea is that let's try to reduce your KV cache um via all the different shenanigans and if you end up with a model you're happy with just serve that. If you're not happy with it, then it at least can be a draft model and you can use your main model to fix things up. Um there's a bunch of um whole literature on speculative uh decoding um right now that improve over the um original u which I'll kind of skip for now.

Okay. So very quickly now um dynamic workloads. Okay. So this is the use case is your you know survey live website and come users come and chat with your model. Um the requests arrive at different times. Um they have different shared prefixes and they have different lengths. Um so it's kind of pretty messy. It's far from this kind of very simple training where you have these blocks of um you know uh the same number of tokens all at once. So what do you do in this case? So there's this um um idea there's a system called Orca that was built. This is actually very early on which uh introduces this idea of continuous uh batching. So the idea is that um you get a bunch of requests that look like this. So here's a prefix of the first request and you're generating this token. Here's a second one. Um here's a third one, a fourth one. It's jagged because every prefix has a different length. And what we're going to do is we're going to decode step by step. So every step you decode one token for all the sequences. Next step you decode another token for all the sequences. And then if you end you just, you know, eject that um se sequence and then as new requests arrive to the batch then you basically put it in the batch and um then you just kind of continue.

Um so that's why it's called continuous batching because it's sort of this your batch is dynamically u being updated um with either old finished uh sequences being evicted and new ones coming in. So now one problem here is that um you know everything we've seen batching works when all the sequences have the same dimensionality right you have tensors everything every every slice has the same dimensionality but in each request here has a different length so what do you do about that so there's this idea called selective batching where um you let's say you have you know a three length three length nine and length five so here um in the tension computation you can't really do anything about this because attention sort of depends on the the length of your your sequence. So if you have a 3x3 you know computation a 9 by9 computation you can't really um you know share the the tensor effectively but for the nonattension the MLP layers which is you know takes up a lot of you know flops you can actually just concatenate all the sequences together to form a mega sequence and you process that okay so final idea is page attention this was introduced in the VLM paper Um, of course, VLM has many other bells and whistles now, but this is kind of the the core idea at that time. So, um, so the question is how is the KV cache stored, right? So, if you think about um requests coming in the you have to put them in uh memory somewhere and in general there's this problem that you get fragmentation. This is what happens to or used to happen to your your your hard drive. Um and you have to defrag your hard drive u you know back in the day. Um so so there's two types of fragmentation.

One is that you have to allocate enough buffer um so that because you don't know a period when you're going to stop. So you might have a max token limit of like you know 1024. So you have to allocate all this memory and you can't put anything in there because you don't you're going to just generate until um you hit the max tokens and that's very wasteful. That's internal fragmentation. And then there's also you have there could be space between different requests and that space is maybe too small to get used effectively. So that's just wasted space. So the solution here is just you know these are systems people so they know their operating systems. They said, "Okay, well, we've solved this problem once before, so let's just use the same idea here." So, we're going to divide the KV cache of a sequence into uh non-ontiguous um you know, blocks. Okay, so if you have this sequence four score and seven years ago, our fathers brought forth, we're just going to chunk it up into these blocks of size four. Um doesn't matter where the blocks go, um but they're going to be kind of aligned according to the blocks. So, there's some uniformity there. Um so when two requests share the same can actually share the same KV cache. Um so um so you might have this block and and this block. So this block might go here and here and this block might be over there. So they're they're in kind of interspersed, but as long as you have the indices and keep track of where everything is, it's fine.

Um so in particular if you have um you know system prompts then you can actually just you know cache these system prompts um the KV cache and the system prompts once right and that can be useful for all the queries okay so this is very useful because if a lot of people are using the same system prompt then you don't have to compute the KV cache for every you know request also there are many applications where you have the same um prompt and you actually want to generate multiple responses. So in that case um you can also just share the KV cache for all the the pro for the prompt and just have unique responses coming out.

Okay. So um so for example if you were to generate let's say multiple um generations from four score and seven years ago R blank and so u what would happen here is that you would have four score and seven and you would start by having years ago or hour and then um this is called uh copy on write semantics. you keep this and then

we have these two samples and if they had happen to sample the let's say the same token you just you know just continue with that but if they sample different tokens you split the block and then you can kind of continue there. So you're basically sharing as much c of the prefix cache as you as you can.

Um there's a bunch of other optimizations like you know uh kernels um that I'm not going to have time to go over. Um but the general idea is that you're using kind of these operating you know systems metaphors to manage um kind of your inference. Okay so summary here inference is really really important. um it's very different from training even though it's the same model but you're asking the model to do something very different ends up being very memory bound and um it's also you know dynamic if you're are kind of in a live chatbot use case we saw a variety of different techniques um to improve inference you can quantize you can come up with new architectures you can prune and distill you can also use speculative um you know sampling but all of these are driven apply the same you know principle here which is reduce your KV cache but don't hurt accuracy too much um and then there's ideas from you know you know uh systems um like paging and speculative execution that can be brought to bear for actually live kind of inference servers um one thing we didn't really get a chance to talk about which was discussed briefly is that I think that new architectures have actually a huge potential for you know improvement things like you know state space models or linear attention or diffusion um these can you know at some level the KV cache and the way that attention is is built fundamentally makes it an inference unfriendly arch uh kind of architecture. So if you can come up with a new architecture that is sort of designed for inference in the way that the transformer was not this can maybe unlock a lot. Okay. So I will uh stop there and next class uh Tatu will return and talk about uh SC laws part two.