[SQUEAKING]

[RUSTLING]

[CLICKING]

SPEAKER: Thanks for having me here today, and also thanks for accepting my request for presenting on this topic. So just for some context, shamelessly reintroducing myself, I work as a research engineer at Hugging Face I try to focus on different facets of diffusion models for image and video generation.

Part of my time at Hugging Face is focused on maintaining and growing the diffusers library, but this talk is not going to be about the diffusers library, and the rest of my time at Hugging Face is also spent around researching diffusion models from many different perspectives. And I'm going to be covering some of that in this talk. As this talk suggests, it's going to be about optimizing the full stack for image and video generation models. In particular, we will see how the optimization aspects of these models are a little nontrivial and also kind of atypical in nature, and also some perspectives into how the whole optimization landscape can transcend beyond latency and speed.

So with that, I'll start. I'll try to also give you a very hand-wavy introduction to diffusion models because diffusion model itself will take another lecture to cover. So I'll try to give you just enough context so that we can step through the rest of the discussion for this talk, and hopefully, by the end of this talk, there will be time for some Q&A. In case there isn't, you can always reach out to me via email, and we can always discuss things offline.

I want to get the excitement rolling by giving you some outputs from some of the recent text-to-image models, but this is from PixArt-Alpha. This is from back in the days in 2024-- I think 2023, not even 2024. So this is quite old, but you can see the quality is not bad. This is from DALL-E 3, OpenAI, and this is from 2024 September if I remember correctly. This is from Flux. And this example is by far my most favorite because even imagining a tiny astronaut, let alone trying to imagine it hatching from an egg, is wild. It's simply wild. But somehow, these models are expressive enough to be able to come up with something as real as this.

There's an ongoing emergence around text-to-video models and the whole moniker around world models as well. This is one of those. And then we have got this cool cat. And then you have got very cinematic light landscape frames. All of these are open models except for DALL-E 3 that I've shown in the previous slide.

As a cautionary note, I'm going to be interchangeably using diffusion and flow in this talk, which also means that the points and the discussions that we'll do in this talk, they will apply to both diffusion and flow models. Flow model-- I like to explain them is a generalization of diffusion models. But they are not exactly the same. But for the purposes of this talk, I'll interchangeably use them.

Now let me try to give you some context as to how diffusion models work. I like to think of diffusion models as the following. What happens when we start from a random noise drawn from a Gaussian? And we then try to slowly denoise it over a period of time unless we get something realistic. It can be an image. It can be a video. Or it can be even a piece of audio as well.

But the crucial point is we are starting from a pure random noise, and then we are iterating over a period of time, denoising it, so that it becomes a clean and realistic representation of what we want, so it can be considered as some of an iterative denoising process. And we can also condition this denoising process. When we try to condition with text descriptions, we can do tasks like text-to-image like we are seeing here.

Now the thing is, diffusion models can have different variants, like when we are operating directly on the pixel space, that family of model is typically referred to as pixel space diffusion model. But pixel space diffusion is pretty intensive from both memory and compute standpoints, which is exactly why operating on the latent space is almost the de facto for diffusion models. And in this diagram, we can get a sense of how latent space diffusion works.

Now we need to have some of an encoder and a decoder in order to be able to encode the pixel representation of an image into its latent representation. And then when we are converting that latent space into the pixel space, we need to have some decoder. Or typically, for the case of latent space diffusion models, we usually use a VAE, which has an encoder variant as well as a decoder variant. So yeah, just note that this talk is mostly going to be about latent space diffusion models. But the approaches are fairly general enough. They will also apply equally. They should apply equally to pixel space diffusion models as well.

Now, let's try to also see how different components within a diffusion model are connected with one another because unlike language models or visual language models, diffusion, any modern state of that diffusion model is not a single model. That's a very important distinction to be aware of.

Now let's say we want to take this text prompt, and we want to generate a video out of it. Now let's see what are the typical components that are involved and how they are connected. Now, to be able to have salient representations of the textual description, we need to have text encoder, which is fairly intuitive. And then once we have the text embeddings out, we start the-- this is the inference workflow, by the way.

Now, with the text embeddings out, we start off with a pure random noise as I was mentioning. These are our noisy latents as we are operating on the latent space, and we also need to have access to a component we term as scheduler, which is a nonparametric component, which takes care of-- which basically makes the model aware of the position it is in the entire denoising trajectory.

And this is the beast that we will try to optimize. But more on that later. It can be a typical unit-like architecture or a transformer-like architecture. And broadly, this is referred to as the diffusion network, which is conditioned on the time step, like the position in the denoising iterations that it is in, the text embeddings in this case, and also the noisy latents. And then we invoke it over a period of time, hence the loop.

Now, once we get our refined legends out the diffusion network, it's passed off to a decoder, and we finally get our frames. In case of an image, it will just be a single frame or just the single final image.

Now, now that we have some very basic understanding of the different components that are involved in a standard text-to-image or text-to-video diffusion model, and how those components are connected with one another, how they draw the chronology in the whole inference workflow. I'll try to now motivate why we might have to optimize them to be able to make anything useful out of them.

Now let's try to look at the memory footprint of the individual model-level components in a fairly state of the art and recent model called Flux. It uses two text encoders, which is also kind of common in the literature of text-to-image diffusion models. For Flux, it uses two text encoders. It has got a T5-XXL, and it has got a clip large. If you are interested in knowing why we need different text encoders and why that might be beneficial, we can talk about it later. But in the interest of time, I'll just keep going. But that's an interesting question to ask. And then you have got the transformer, which is like the diffusion network. And as we can see, it has got the highest footprint. And you have got the decoder, which will be responsible for taking the defined latents out of the transformer and then decoding it back to the pixel space.

And as I was mentioning, the transformer or the diffusion network here is the most compute intensive unit. And it's compute bound, unlike language models. Hence, consequently, the optimization literature from the language modeling world might not carry over to the diffusion world very gracefully.

Now, even when using the brain float16, which is very common in the diffusion world, it takes about 34 gigs to generate a 1024 by 1024 resolution image, which is standard, and it takes about 7 seconds on a fairly beefy GPU such as H100, without any optimizations. So I hope you we can see how these numbers are kind of staggeringly high because if we have to wait for about 7 seconds to generate one single image, it's not good. And then videos can be even worse, like a fight. So just to give you some perspective, a 5 second, 16 FPS, 720p video can take about 30 minutes to generate fairly state-of-the-art open video generation model.

Now these numbers are heavy. These numbers should concern us in case we are interested in operationalizing some of these models because long-running tasks are bad for user-facing applications. It hinders interactions because if the interactivity aspect of creative applications is hindered, it's not good for your users, and it's also power inefficient. And it also slows down the overall rate at which you would want to iterate and make some improvements to these models. So it's kind of a bad experience for both the developers as well as the users of these models.

Now, as I was hinting at the iterative transformer. It's at the root of all evil here, basically, because it's both compute bound and same time very memory hungry now and a natural question because the transformer takes the most amount of time in the whole generation workflow. A natural question here to ask would be do we just optimize for speed?

But what happens when you also try to motivate the entire optimization landscape with an application, the application in which this diffusion network is going to get used? If we consider things from that perspective. There will be other factors that will influence the whole process, like the use case in which this network is used, the level of user interaction, the rapidity of the user interaction, the kind of throughput we will have to meet, and also the deployment hardware that's available to us.

So I think I was able to motivate why you would want to work with an optimized version of the diffusion network, and also why speed might not just be the only factor that we want to care about in this case. Hence, I am going to talk a little more about how optimization becomes a factor that transcends well beyond just speed.

Effectively and essentially, I like to think of the whole optimization landscape as a few guiding principles, when should, when should the optimization take place? And in this case, the motivation basically comes from the app where should we perform optimization. And in this case, it's roughly going to be the diffusion network, the transformer model. That's the most compute-intensive element in our workflow.

And do we know what we want to optimize? Do we want to optimize throughput? Do we want to optimize memory? Or do we want to optimize both? And how should we optimize? This is where this talk is going to be focusing on. So yeah, if we are going to be discussing a couple of different recipes and approaches as to how we can optimize the transformer network and also how we can go beyond that.

Now I think there needs to be more motivation and more awareness to be put when it comes to the kind of hardware that we have access to especially when we are deploying these models. Now in this picture, we can see how the throughput immediately improves when we try to use shapes that are particularly optimized for a given hardware.

Now, on the extreme right-hand side, the bar shows, it's basically configured with the most optimized shapes for a given hardware, and it gives us the most amount of throughput. And the number of total model parameters is not changing. That's not changing. We are just using the shapes within the model, maybe, the number of attention heads or maybe the hidden dimension of the transformer blocks and so on. And it shows how just changing that and how optimizing that with respect to the given hardware can immediately render some positive gains in terms of throughput.

So that's what I was mentioning. Nontrivial gains can come from kernels that are hyper specialized for a given hardware, and also model architectures where the shapes of the model are kind of optimized and have been made efficient with respect to the hardware. It's going to be operationalized.

Now there's also this aspect of efficiency that's going to keep coming up because we want to make our model as efficient as possible. But efficiency is also often a term that has so many misnomers. The popular belief is smaller models are almost always faster and more efficient. The reality is no, not always. And I have this very popular figure from this paper called "the efficiency misnomer" as the title of my slide goes. It basically shows how models with higher number of parameters may not have the highest number of flops, and also it may not have a lower throughput.

Now, in this case, it's basically reversed. That is, the models with a lower number of parameters has more flops and has a lower amount of throughput than their apparently heavier counterparts. So that kind of drives this point home. That efficiency may not always be related to models being smaller. There are more angles to take a look at it.

Now the big elephant is also transformer. I'm going to keep coming back to this transformer thing. But also, when it comes to high resolution generation of images and videos, the effects of the compute-bounded regime of our transformer can become really, really evident. First, the inputs are of high dimensional. For example, for 4K image generation, even if we were to operate with a compression factor of 8x, we have got this huge dimensionality problem. We have got batch size. Then we have nominating channels, which can be 64, 128, and so on. And then you have got your latent width and latent-- you have got your latent height and latent width, which is also like 512 and 512.

Now doing attention at this high-dimensional space can immediately restrict-- can immediately impose memory implications as well as speed implications. Two very popular ways to deal with this high-dimensionality problem, in order to achieve some efficiency gains, is we increase the compression factor, like from 8x, we compress even higher, maybe 16x or 24x, and we try to recover the lost information through other means. There are several works that have explored this, but I just wanted to give you a high-level overview of the typical things that are done when we have this problem of high-dimensional representation spaces.

Now when we-- just to elaborate a bit further on this point, when we try to compress higher and higher, we also end up losing a lot of information redundancy, which might be actually necessary study in order to model the expressivity into the overall workflow. And if we do miss out on that, our quality can take a huge hit. Now, that's also why it's essential to try to recover the information loss incurred during this heavy compression, so some kinds of other means.

And as a motivating example, I want to show you this picture from this very popular high resolution image synthesis model called SANA, which also happens to operate on a highly compressed latent space. And as we can see, using extreme compression can have significant gains on the speed. So in this case, we can see a 25x reduction in the generation latency when operating with 4K images.

Now, the good thing about trying to do this more from a first principles approach is these approaches should be complemented with the kinds of techniques that we have for latency optimization, for example, maybe an optimized kernel or maybe using things like FlashAttention.

So at this point in time, I would expect us to have a good understanding of how we can optimize the model architecture and also have a model architecture that respects the hardware that it will get deployed to. Now, ultimately, since we are also focusing on the applications in which these models are going to be used, I think it also makes sense to try to think about how we can also optimize a little bit for the use case. So let's see how.

So we know that my model is topping all the standard generation benchmarks. That's all well and good. But the use cases in which the model is going to get deployed to, they might be different. For example, maybe the model would be needed for photorealism. So in that case, the users would demand for specific attributes to be better than others. And then maybe the model will be used in the context of interactive generation. In those cases, we would rather have real-time interaction and lowest possible latency. And maybe quality doesn't matter that much.

So in these cases, we will have to finetune whatever model that we have for specific needs. And that entire process should be driven by the use cases. Now one such way to do that, one such way to inject some form of use case awareness would be to do preference alignment, where the model is finetuned to output what's preferred. In this case, you have got a text prompt, a cyberpunk cat with a neon sign that says Sana, and you have got two images, two candidates. Now, the model will be trained on this triplet, and it will be basically trained to output what the users typically prefer.

We might also want to extend the way users prompt these models. So far, we have been seeing text-to-image models, which is basically users provide natural text description of what they want to see in the final outputs. Maybe that's not sufficient. Maybe I also want the output image to follow a particular pose. Maybe I want to condition the model with more structural forms of inputs such as pose, segmentation, map, Canny map for example. And in this case, I'm basically prompting the model with a certain pose as well as a textual description. And as you can see, it works, basically. So maybe the textual prompt was not enough. And hence there was a need to allow the users to have more control over what they can provide to the model as inputs.

If training is not possible because training is an expensive gig, even with parameter efficient finetuning, the results might be far from expected. In those cases, we can also incorporate things like inference time scaling, where we might want to search for better noise. As we saw a couple slides back, when performing inference with diffusion models, we try to start from a random noise, and then we try to denoise it. We could search for better noise, like the initial noise could be better because not all noises are equal. They are not going to lead to same quality outputs. So we might want to search for better noise, or maybe we want to search for a better prompt. And it's also possible to search for both, actually.

Now this gives us a depiction of what I wanted to convey. So you basically generate an image from a language prompt. And then you compute some metric. In this case, it's the ClipScore. And then if the metric is OK, then all good. Our job is done. If the metric is not OK, maybe we ask some language model to come up with a better prompt. And then we enter into this interactive scaling framework. Now Clip Score here I used clip score as a motivating example. But this metric should be use case-specific.

So this is the last section of my talk, where I want to give you a flavor of some of the more advanced optimization techniques that we can incorporate. So knowledge distillation is already very popular. We have had many papers in the community that talk about this technique. So the basic idea is we have a larger model that is probably memory intensive and also slower and so on. And we want to distill that large model into a smaller one, which is often known as some of a compressed representation of that larger model.

And in the case of diffusion models, we have got large number of iterations. And we can typically reduce the number of steps or iterations needed to generate something reasonable. And in the literature of diffusion models, it's known as time step distillation. And also, one advantage of working with diffusion models is that we can combine architectural compression with time step distillation and come to be able to benefit from the best of both worlds.

So at a glance, if I were to give you an overview of the things that I find to be beneficial when trying to approach the optimization landscape for diffusion or flow models is we should incorporate hardware awareness when designing the model architecture and also shapes that are optimized for a given hardware for potentially better efficiency. And it also can be very beneficial to make the model architecture flexible, to be able to extrapolate to use cases like high resolution image and video synthesis.

And we can complement the benefits that come from model architectures with latency optimization techniques such as better kernels, FlashAttention, and those techniques. And if the use cases demand for it, we should post train. And if latency requirements, demand for it, we might want to just distill as well. So I think that's about it. That went quicker than I was expecting. And most of the content of this talk I have written in this blog post. So yeah, that's about it from my end. So if there's any questions, I'm happy to take them. So yeah.

HOST: Awesome, thank you so much.