60-second takeaway
FP8 on an RTX 3090 Ti is real, but it is mostly a **VRAM-saving storage trick**, not an FP8 acceleration path.
The 3090 Ti is an Ampere GPU with compute capability 8.6. It can hold weights in `torch.float8_e4m3fn`, but native FP8 tensor-core compute is not the path you should count on. The reliable consumer-GPU recipe is to store large diffusion transformer weights in FP8, then compute in BF16.
If a model almost fits in 24 GB, use diffusers layerwise casting first. If it still does not fit, combine FP8 storage on the diffusion transformer with NF4 on the text encoder.

Who this is for

This guide is for builders running image or video generation models on a single 24 GB NVIDIA GPU:

- RTX 3090
- RTX 3090 Ti
- A10
- RTX 4090
- L40 / L40S

The most common reader has a model that almost fits. A README, issue comment, or benchmark says "use FP8", but the same recipe was probably written on a 4090, L40, H100, or newer card. On a 3090 Ti, that detail matters.

The question is not "does PyTorch expose FP8 dtypes?" It does. The question is:

Which FP8 paths actually help on Ampere, and which ones quietly assume newer hardware?

The short answer

On RTX 3090 Ti:

```
pipe.transformer.enable_layerwise_casting(
    storage_dtype=torch.float8_e4m3fn,
```

```
    compute_dtype=torch.bfloat16,
)
```

This is the practical path. It stores transformer weights in FP8, then casts them back to BF16 when each layer runs.

Do not expect this to make generation faster. Expect it to cut weight memory enough that a larger model or higher resolution run fits.

What SM 8.6 vs SM 8.9 means

SM means Streaming Multiprocessor architecture version. In CUDA docs, this is usually described as compute capability.

The useful boundary for this post:

GPU family	Example GPUs	Compute capability	Practical FP8 meaning
Ampere consumer / prosumer	RTX 3090, RTX 3090 Ti, A10	8.6	FP8 storage is useful; compute should stay BF16 or FP16
Ada	RTX 4090, L4, L40, RTX 6000 Ada	8.9	FP8 tensor-core paths become hardware-relevant
Hopper	H100	9.0	FP8 is a first-class datacenter training/inference path

NVIDIA's CUDA tuning guide identifies Ampere as compute capability 8.0 and 8.6, and Ada as compute capability 8.9. NVIDIA's Ada architecture materials also call out fourth-generation Tensor Cores with FP8 support, while Ampere materials emphasize TF32, BF16, FP16, INT8, and INT4 rather than FP8.

That is the boundary. The 3090 Ti can be very useful for local AI production, but it is on the wrong side of the native FP8 compute line.

Why this confuses people

There are three different ideas hiding under the word "FP8":

Phrase	What it means	3090 Ti result

FP8 dtype exists	PyTorch can represent tensors with a float8 dtype	True
FP8 storage	Weights are stored in FP8 and cast up for compute	Useful
FP8 compute	Matrix multiply uses FP8 tensor-core kernels	Not the reliable Ampere path

Most blog posts and repo comments blur these together. That is how people end up trying FP8 activation quantization recipes on a 3090 Ti, then wondering why the result fails, falls back, or produces poor output.

The rule for Ampere is simple:

Use FP8 to store weights. Use BF16 or FP16 to compute.

The recipe that worked on RTX 3090 Ti

The cleanest path is diffusers layerwise casting. It is designed exactly for this use case: keep module weights in a low-memory storage dtype, then upcast layer by layer during the forward pass.

```
import torch
from diffusers import DiffusionPipeline

pipe = DiffusionPipeline.from_pretrained(
    "your/model",
    torch_dtype=torch.bfloat16,
)

pipe.transformer.enable_layerwise_casting(
    storage_dtype=torch.float8_e4m3fn,
    compute_dtype=torch.bfloat16,
)

pipe.transformer.to("cuda")
pipe.vae.to("cuda")
```

On a 9B diffusion transformer, the mental model is:

Component	BF16 memory	FP8 storage memory
9B transformer weights	~18 GB	~9 GB

That memory reduction is the win. The forward pass still uses BF16 compute, so you should not sell this to yourself as an acceleration trick.

The 24 GB diffusion pipeline pattern

For modern image and video models, the transformer is not the only large component. Text encoders can be huge.

The pattern that gives the most practical headroom on a 24 GB card is:

- Diffusion transformer: FP8 layerwise casting
- Text encoder: NF4 via bitsandbytes
- VAE: BF16
- CPU offload: only after you have measured the real component sizes

Example for a FLUX-style pipeline:

```
import torch
from transformers import BitsAndBytesConfig as TransformersBitsAndBytesConfig

text_encoder_config = TransformersBitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True,
)

text_encoder = TextEncoderClass.from_pretrained(
    repo_id,
    subfolder="text_encoder",
    quantization_config=text_encoder_config,
    torch_dtype=torch.bfloat16,
    device_map="auto",
)

pipe = PipelineClass.from_pretrained(
    repo_id,
    text_encoder=text_encoder,
    torch_dtype=torch.bfloat16,
)

pipe.transformer.enable_layerwise_casting(
    storage_dtype=torch.float8_e4m3fn,
    compute_dtype=torch.bfloat16,
)
```

For a 9B transformer plus an 8B text encoder, the rough 3090 Ti budget looks like this:

Component	Technique	Approx memory
Diffusion transformer	FP8 storage, BF16 compute	~9 GB
Text encoder	NF4 double quant	~4 GB
VAE	BF16	~0.2 GB
Activations and overhead	depends on resolution	~2-3 GB
Total		**~15-16 GB**

That leaves real headroom on a 24 GB card.

What not to do first

1. Do not start with FP8 activation quantization

Activation quantization is a compute-path feature, not just a storage feature. This is where the hardware boundary bites.

If you are on a 3090 Ti, start with layerwise casting. Keep the compute dtype BF16 or FP16.

2. Do not assume a single-image smoke test proves the batch run

We have seen this pattern repeatedly:

1. One image passes.
2. The second or third image OOMs.
3. The issue is not the model "not fitting" in the simple sense.
4. The issue is component swapping, allocator fragmentation, or repeated-pass overhead.

If your run depends on CPU offload, validate with a multi-image batch. A clean first forward pass is not enough.

3. Do not assume a pre-quantized FP8 repo is a diffusers pipeline

Some FP8 model repos ship as a single `safetensors` file. That can be useful, but it is not the same as a complete diffusers repo with `model_index.json`, scheduler config,

VAE config, and component subfolders.

The safer path is:

1. Load the complete BF16 or standard diffusers repo.
2. Apply quantization on load.
3. Save your working loader script.

Use a single-file FP8 checkpoint only when you are ready to handle conversion and state-dict loading details.

Technique comparison on RTX 3090 Ti

Technique	Memory effect	Speed effect on 3090 Ti	Quality risk	Use when
BF16 baseline	Highest memory	Reference	Lowest	Model already fits
FP8 layerwise casting	Cuts weight memory ~50%	Usually not faster	Low	Model almost fits
torchao float8 weight-only	Similar memory goal	Usually not faster	Low to medium	You need torchao-specific flow
NF4 bitsandbytes	Cuts memory more aggressively	Can be slower	Medium	Text encoder is too large
INT8 weight-only	Moderate memory reduction	Usually not faster	Low to medium	FP8 path is awkward or unsupported
CPU offload	Reduces peak GPU residency	Slower	Low	Components fit one at a time

Default recommendation:

1. Try BF16 first if it fits.
2. If it almost fits, apply FP8 layerwise casting to the transformer.
3. If the text encoder is the next bottleneck, use NF4 for that component.
4. If total component size exceeds VRAM by a lot, add CPU offload and test more than one generation.

A practical decision tree

If the model fits in BF16

Do not quantize by default. You will save engineering time and avoid surprising quality drift.

If the transformer alone is too large

Use FP8 layerwise casting:

```
pipe.transformer.enable_layerwise_casting(
   storage_dtype=torch.float8_e4m3fn,
   compute_dtype=torch.bfloat16,
)
```

This is the best first move on a 3090 Ti.

If the text encoder is too large

Use transformers.BitsAndBytesConfig for the text encoder. Keep this separate from diffusers quantization config classes.

```
from transformers import BitsAndBytesConfig
```

The text encoder is usually a better NF4 target than the diffusion transformer because a small quality loss in the text embedding path is often less visible than quantization damage in the denoising transformer.

If both components individually fit, but the pipeline OOMs

Use component-level offload and validate with repeated generations. The trap is thinking "each component fits" means "the whole run is stable." It often does not.

If a README says "use FP8 for speed"

Check the GPU used in that benchmark. If it was Ada, Hopper, or Blackwell, do not transfer the speed claim to Ampere.

The two BitsAndBytesConfig classes trap

There are two similarly named config classes:

Component	Correct config class

Transformers text encoder	transformers.BitsAndBytesConfig
Diffusers model component	diffusers.BitsAndBytesConfig

They have overlapping parameter names, but they are not interchangeable. Mixing them is a fast way to get confusing loader errors.

For most 3090 Ti diffusion work, avoid this trap by using:

- transformers.BitsAndBytesConfig for the text encoder
- enable_layerwise_casting() for the diffusion transformer

That keeps the two worlds clean.

What we validated

Our local validation was on an RTX 3090 Ti with 24 GB VRAM.

The durable findings:

- FP8 layerwise casting is useful for large diffusion transformers.
- On Ampere, the value is VRAM reduction, not FP8 compute acceleration.
- A 9B transformer can become practical in a 24 GB pipeline when stored in FP8.
- Combining FP8 transformer storage with NF4 text encoder quantization creates enough headroom for FLUX-style pipelines.
- CPU offload can pass a smoke test and still fail in repeated runs if the component budget is too tight.

That is the part worth taking into production.

Sources

- NVIDIA CUDA Ada tuning guide: Ada devices are compute capability 8.9
- NVIDIA Ampere architecture page: Ampere Tensor Cores support TF32, BF16, INT8, and INT4
- NVIDIA Ada architecture page: Ada Tensor Cores add FP8 precision
- NVIDIA TensorRT for RTX support matrix: compute capability 8.6 lists FP8 as not supported
- Hugging Face diffusers memory docs: layerwise casting with torch.float8_e4m3fn storage and BF16 compute

- Hugging Face diffusers model docs: model-level enable_layerwise_casting
- PyTorch torchao inference docs: float8 dynamic activation + float8 weight workflow requirements

Related Instavar posts

- Voice Cloning on a 24GB GPU - What Actually Works in 2026
- How to Run an AI Video Model Bakeoff Without Turning It Into Vibes
- What a Production-Grade AI Video Pipeline Actually Needs
- HunyuanVideo 1.5 - Upgrade Checklist for Production Teams

The honest read

FP8 on a 3090 Ti is not fake. It is just narrower than the marketing phrase suggests.

Use it to make weights smaller. Do not expect Ampere to behave like Ada or Hopper. If you keep that boundary clear, FP8 storage is one of the simplest ways to make large image and video generation models practical on hardware many indie builders already own.