# End Term Report
# Team AIML-25

**Team Leader:**
Divyansh Gupta

**Team Members:**
Keshav Lakshmi Narasimhan |Rishan Gobse |Kartik Budhani |Krishnam Digga

**Mentor:**
Harshvardhan Chaudhary

August 2nd, 2025

## 1 Introduction:

- We are attempting the eighth problem statement of IITI SoC '25 in AI - ML domain. The PS is: Accelerate - WAN – Optimizing Inference and Training Speed of the WAN 1.3B Video Model.
- We targeted the WAN 1.3B T2V model, which is the state-of-the-art model for text-to-video generation.
- Our general approach was to first load the baseline model, then establish a pipeline that is able to generate the video, using the model by giving appropriate inputs, and measure the appropriate metrics, and then finally test and implement optimization techniques.
- Until now, we have successfully loaded the model both using diffusers and cloning the repo (locally). We have measured various metrics as well as implemented the following optimization techniques on the model loaded using diffuser - operator fusion, LoRA and Quantization.

## Contents

## 2 The Diffusion Algorithm

Diffusion is an algorithm for image and video generation. It relies on the concept of taking a training image and gradually adding gaussian noise until a perfect normal (gaussian) distribution is reached. Then, a neural network is used to gradually remove the noise, or denoise, until a clear output image is reached. The model does this in several steps, known as inference steps.

### 2.1 Gaussian Noise

Gaussian noise is a random noise instance from a normal distribution. The normal distribution is a bell-shaped curve that maps values to their probability densities. A vector/matrix of random values selected from this distribution can be chosen depending on the required dimensions, which we will call e. The equation of a normal distribution is:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Where $\mu$ corresponds to the mean and $\sigma^2$ corresponds to the variance.

### 2.2 Training

The process is initiated with a clean image $x_0$ and a random Gaussian noise vector $\epsilon$. Each step of adding or removing noise is known as a time step. A random time step $t$ is also chosen. Using these three variables, a noisy image is calculated by throwing in a certain amount of noise based on $\epsilon$ and $t$, producing $x_t$. The formula for $x_t$ involves blending $x_0$ and $\epsilon$ in a certain proportion to produce a noisy image.

If 1000 time steps were used, $t = 0$ would correspond to the clean image and $t = 1000$ would correspond to $\epsilon$ itself, pure noise. From the previous noisy image $x_{t-1}$, we can derive the current noisy image $x_t$ by applying the forward process shown below.

$$x_t = \sqrt{1-\beta_t}\, x_{t-1} + \sqrt{\beta_t}\, \epsilon$$

This relies on blending the current noisy image with more Gaussian noise to produce the next noisy image. The "weight" that controls this mixture, $\beta_t$, varies with timestep. $\beta_t$ is a function of $t$. This function is known as the schedule.

For training, $x_t$ must be directly represented as a function of $x_0$, the original image, and not $x_{t-1}$. For this, an additional parameter $\alpha_t$ is defined such that $\alpha_t = 1 - \beta_t$. So, the equation becomes:

$$x_t = \sqrt{\alpha_t}\, x_{t-1} + \sqrt{1-\alpha_t}\, \epsilon_t$$

In order to obtain an expression in $x_0$, we substitute $x_{t-1}$ in terms of $x_{t-2}$, then $x_{t-2}$ in terms of $x_{t-3}$, and so on. Once this recursive substitution is finished, the following equation is obtained:

$$x_t = \sqrt{\bar{\alpha}_t} \cdot x_0 + \sqrt{1-\bar{\alpha}_t} \cdot \epsilon$$

Here, $\bar{\alpha}_t$ is the product of all values of $\alpha$ across all time steps until the current time step.

$$\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$$

Note that $t$ can only take integer values, so this is not an infinite multiplication. $\alpha$ is initialized just below 1 and slowly decays toward 0. Since all $\alpha$ values are below 1, $\bar{\alpha}_t$ decays to 0 much more rapidly as time steps pass.

When $t = 0$, $\bar{\alpha}_t$ is near 1 but not exactly 1. This means that $x_0$ is not exactly the input image, and we start with a tiny bit of noise. As $t$ increases, $\bar{\alpha}_t$ tends to 0, which means $x_t$ tends to $\epsilon$.

Once a random noise vector $\epsilon$ is generated, a random timestep is chosen and $x_t$ is calculated, the model is given $t$ and $x_t$ and is trained to independently predict the noise vector to be added to $x_0$ to produce $x_t$.

## 2.3 Inference

After the model learns to add noise to clean images, it is asked to do the reverse during inference. Let 1000 time steps be used. The model is provided with pure Gaussian noise along with its corresponding time step ($t = 1000$) and is tasked with resurrecting a clean image. Essentially, given pure noise, it is asked to predict the vector $\hat{\epsilon}_t$ (not to be confused with the previous definition) which, if subtracted from the pure noise, will give a clean image $x_0$. The model can directly subtract this vector and obtain a clean image in one shot, but this will result in inaccuracy. This is because small errors that arise while predicting $\hat{\epsilon}_t$ will shoot up significantly. To understand this, consider the rearranged equation:

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\cdot\hat{\epsilon}_t}{\sqrt{\bar{\alpha}_t}}$$

As $\bar{\alpha}_t$ is very close to 0 at the maximum timestep, even small errors in $\hat{\epsilon}$ will cause a massive change in the pixels of $x_0$. This will almost certainly result in a highly skewed image.

Instead, the predicted value of $\hat{\epsilon}$ and $x_{1000}$ is used **to directly calculate the output of the previous timestep**, i.e. $x_{999}$. Then, the model is fed this new noise $x_{999}$ along with its new timestep ($t = 999$) and is asked to predict another $\hat{\epsilon}_{999}$, which we use to reconstruct $x_{998}$. We repeat this process until we reach $x_0$, the clean image. Note that $t$, the total number of time steps, can be adjusted as necessary.

## 2.4 VAE and U-Nets

Diffusion implements VAE and a modified version of the U-Net architecture to guide the model to generate desirable images. The U-Net is originally a CNN based architecture used in biomedical image segmentation. A cross-attention mechanism was added to allow a text prompt to condition the diffusion process. The entire process explained in the following is repeated whenever the model is to predict the vector $\hat{\epsilon}_t$ given $x_t$.

The input is an RGB image, say 3 x 512 x 512 (3 RGB channels). Before entering the U-Net, the image is passed into a VAE encoder (Variational Autoencoder). This encoder uses convolutions to spatially downsize the image, say to 64x64 pixels. However, the output is not merely a smaller RGB image but a latent tensor with 4 channels. These channels do not represent color, but feature representations such as edges, shapes and texture. Hence, a 4 dimensional vector is associated with each spatial location. While Stable Diffusion v1 uses 4 channels, this number can be varied. A very low number will result in poor feature recognition (underfitting) and too high a number will consume resources and cause overfitting. Therefore, in this scenario, the latent tensor has dimensions of 4 x 64 x 64.

Once the image has been converted into a latent, it is passed into the encoder of the U-Net. This encoder, not to be confused with the VAE encoder, downsamples the image further. The current timestep is also embedded and incorporated in multiple locations in the U-Net. Attention blocks are inserted at various resolutions. The mechanism is explained below.

Each of the 4-dimensional vectors of the latent tensor is considered a token. The whole vector is thus flattened into 4 x 4096 dimensions. A linear layer within the U-Net projects these vectors into a higher dimensional space, generating embeddings used in attention. The given text prompt is also tokenized and converted into embeddings with the help of CLIP. Key vectors and value vectors are then calculated with the learnable weight matrices. A cross attention mechanism is then applied with the query vectors of the image tokens and key & value vectors of the text prompt. An output vector is then generated with the attention scores.

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

The image is at its lowest spatial resolution at the bottleneck of the U-Net. Next, the latent enters the decoder which gradually scales up the image. Skip connections between corresponding layers of the encoder and decoder are applied to restore the features extracted in the encoder, should they have been degraded in the downsampling process. This is aided by the fact that the encoder and decoder are perfectly symmetric in their structures.

The output of the U-Net is a latent tensor of the same dimensions as the input: 4 x 64 x 64. Finally, this tensor is passed to a VAE decoder which brings it back into the RGB space, resulting in a 3 x 512 x 512 image. It is important to note that this is not $x_{t-1}$, but the predicted noise vector $\hat{\epsilon}_t$. $x_{t-1}$ is then calculated by subtracting a portion of this noise vector $\hat{\epsilon}_t$ from $x_t$. This entire process is iterated over all timesteps until all noise has been removed and a clean image is obtained.

## 3 Setting up the pipeline for base model:

- To load the model, two methods were possible - using diffusers library or cloning the model repository. We decided to explore both.
- First we loaded the model with diffusers library. Then we designed a basic function that takes prompt, fps, frames, etc as input and saves the output video and frames as a .mp4 file at specified path.
- We also loaded it locally by cloning it's repo. We followed the instructions given in README file of the model repo to load it and to generate the output.

## 4 Setting up the evaluation system:

- Next, we started exploring the metrics - other than the ones specified in PS, for highlighting the effects of our optimizations. Finally, we selected following metrics:

### 4.1 Efficiency Metrics:

#### 4.1.1 Time vs. Denoising Steps Graph:

This graph illustrates how much time is consumed during each of the 50 inference steps in the diffusion-based video generation process. Rather than making changes to the model's internal code, we opted for a simpler and non-intrusive method: we modified the tqdm progress bar — which is typically used to display real-time progress during iteration — to record a timestamp at each individual step. These timestamps were then collected and used post-process to compute the time difference between steps. This allowed us to generate a detailed graph that reflects the variation in computational load across the denoising steps, helping us better understand the runtime characteristics of the diffusion pipeline.**Clip-wise and Frame-wise Latency:**Clip-wise latency refers to the total time taken to generate an entire video clip, starting from the moment the generation process begins until the final frame is produced. This metric captures the overall end-to-end time required for a single video output. Frame-wise latency, on the other hand, provides a finer-grained view by calculating the average time taken to generate each individual frame. It is computed by dividing the total clip-wise latency by the number of frames in the video. This is especially useful for benchmarking performance across different video lengths or comparing models that generate varying numbers of frames. Both metrics together offer a comprehensive view of the model's generation speed and are essential for evaluating performance in time-sensitive or resource-constrained scenarios.

- – **How it's done:**
  1. A timer is started at the beginning of the video generation process.
  2. The timer stops once the last frame of the clip is generated.
  3. The total elapsed time is recorded as the **Clip-wise Latency**.
  4. This value is then divided by the total number of frames to obtain the **Frame-wise Latency**.

These latency metrics help benchmark model efficiency and compare different optimization settings, hardware platforms, or generation configurations.

$$Clip = tend - t_{start}$$

**Where:**

- $t_{start}$: Time when generation begins for a clip
- $t_{end}$: Time when the last frame is generated or output is finalized
- Typically measured in **seconds per clip**

$$Frame = \frac{ttotal}{N}$$

**Where:**

- $t_{total}$: Total time to generate the clip
- $N$: Number of frames in the clip
- Measured in **milliseconds or seconds per frame**

#### 4.1.2 Throughput (frames/sec):

Throughput measures the speed of video generation by indicating how many frames the model can produce per second during inference. It serves as a key performance metric for evaluating the efficiency and responsiveness of the generation pipeline, especially in real-time or large-scale applications. Higher throughput implies faster generation, which is beneficial for interactive systems and batch processing alike.

- **How it's done:**
  1. A timer records the total time taken to generate all frames in a clip or batch.
  2. The number of generated frames is counted.
  3. The number of frames is divided by the total time to compute frames per second (FPS).

  Throughput is a key metric for evaluating system-level efficiency, especially when comparing different optimizations, batch sizes, or deployment setups.

$$Throughput = \frac{N}{t_{total}}$$

**Where:**

- $N$: Number of frames generated
- $t_{total}$: Total time taken for generation (in seconds)
- Result is in **frames per second (FPS)**

#### 4.1.3 Warm-up Time:

Warm-up Time refers to the initial delay between when a prompt is submitted and when the actual video generation begins. It captures the overhead associated with tasks that occur before denoising starts—such as tokenizer initialization, scheduler setup, model graph compilation (if applicable), and latent variable preparation. This metric is useful for identifying and optimizing non-denoising bottlenecks in the inference pipeline.

- **How it's done:**
  1. The model is initialized and loaded into GPU or CPU memory.
  2. The first inference is triggered with a sample input.
  3. A timer measures the duration from the beginning of this process to the completion of the first output.

4. This duration is recorded as the warm-up time.

This metric is crucial in real-time or on-demand systems where fast response is needed after model loading.

$$Warm\_Up\_Time = t_{first\_output} - t_{model\_load\_start}$$

**Where:**

* $t_{model\_load\_start}$: Timestamp when model loading begins.
* $t_{first\_output}$: Timestamp when the first output is generated. Measured in seconds or milliseconds.

## 4.2 Accuracy Metrics:

### 4.2.1 LPIPS (Learned Perceptual Image Patch Similarity):

LPIPS is a perceptual similarity metric that measures the visual difference between image pairs using deep neural network features, rather than raw pixel values. In our case, we apply LPIPS to evaluate the consistency between consecutive frames in generated videos. This helps us quantify temporal coherence — that is, how smoothly one frame transitions into the next. A lower LPIPS score indicates high visual similarity between frames, meaning the video appears stable and smooth to the human eye. Conversely, a higher score suggests noticeable visual changes or flickering between frames. The final LPIPS value is computed by averaging the similarity scores across all consecutive frame pairs in the video, making it a reliable indicator of perceptual temporal stability. **Here's how it's done:**Both images (or

frames) are passed through a pretrained network (VGG). Feature maps from selected layers are extracted and normalized. L2 distances are computed between the feature maps of the two images. These distances are weighted and summed to produce the final LPIPS score.

A lower LPIPS score means the images are more perceptually similar in terms of deep visual features.

**Temporal LPIPS:** Temporal LPIPS extends the standard LPIPS metric to videos. It evaluates temporal coherence by comparing how visually consistent adjacent frames are over time.

1. For each pair of consecutive frames in a video, denoted as $(f_t, f_{t+1})$, LPIPS is computed.
2. These pairwise scores are averaged across the entire video sequence to obtain a single temporal coherence score.

$$LPIPS(x, x_0) = \sum_l w_l \cdot \left\| f^l(x) - f^l(x_0) \right\|_2^2$$

$$Temporal = \frac{1}{N-1} \sum t = 1^{N-1} LPIPS(f_t, f_{t+1})$$

**Where:**

* $f^l$: normalized feature map from layer $l$
* $w_l$: learned layer-specific weights
* $x, x_0$: images being compared
* $f_t, f_{t+1}$: consecutive frames in a video
* $N$: total number of frames

### 4.2.2 CLIP Score:

This metric measures how well the generated video semantically aligns with the given input prompt. It captures whether the content of the video reflects the intent, objects, actions, or themes described in the prompt. To evaluate this, we use OpenAI's CLIP (Contrastive Language–Image Pretraining) model, which jointly embeds text and image data into a shared feature space.

* **How it's done:**
  1. The input text prompt is encoded using CLIP's text encoder to generate a text embedding.
  2. Each frame of the generated video is passed through CLIP's image encoder to produce frame-wise visual embeddings.
  3. The cosine similarity between the text embedding and each frame embedding is computed.
  4. These similarities are then averaged across all frames to obtain the final CLIP score.

A higher CLIP score indicates stronger semantic alignment between the video content and the input prompt, meaning the video more accurately represents what was asked for. This makes CLIP a powerful tool for assessing the relevance and faithfulness of generated content in text-to-video tasks.

$$CLIP\_Score(I,T) = \frac{E_I \cdot E_T}{\|E_I\| \cdot \|E_T\|}$$

## 4.3 Hardware Metric:

We used a background thread to collect real-time GPU statistics during video generation. Using `nvidia-smi`, we recorded:

### 4.3.1 Memory Used / Free:

It refers to the amount of unused or available GPU memory (also known as VRAM) at the point just before the inference process starts. This metric is important because it provides an estimate of how much memory headroom is available on the GPU for loading the model, feeding in input data (such as prompts, latent variables, or images), and storing intermediate results during computation. A higher value of free memory suggests more room for heavier models or larger batch sizes, whereas a lower value could indicate a risk of out-of-memory (OOM) errors if the model or input exceeds capacity. Monitoring this value helps in making informed decisions about model size, optimizations, and hardware requirements.**How it works:** Before starting inference, GPU status is checked using monitoring tools like `nvidia-smi`. The amount of free VRAM (not allocated to any process) is recorded at that point in time. This recorded value is then reported as the Memory Free metric. It acts as a baseline for evaluating how much additional memory the model will consume during execution, and helps assess whether the current environment (especially in shared or multi-user systems) can support the model's memory requirements without exceeding limits.

**Memory used**

$$Memory = \max(M_t) \quad for t \in [tstart, t_{end}]$$

**Where:**

* $M_t$: Memory used at time $t$ (observed externally)
* $\max(M_t)$: Peak observed usage during inference
* Measured in **MB** or **GB**

**Memory Free**

$$Memory = Mtotal - M_{used}$$

**Where:**

- $M_{total}$: Total GPU memory available on the device
- $M_{used}$: Memory currently occupied before inference starts
- Measured in **MB** or **GB**

### 4.3.2 GPU Utilization:

Indicates how actively the GPU cores were being used during the inference process. It reflects the percentage of time the GPU was engaged in executing computations as opposed to being idle. A high utilization value suggests that the GPU is being effectively leveraged and working near its full capacity, whereas a low utilization might point to bottlenecks elsewhere in the pipeline — such as CPU-bound operations, memory transfer delays, or inefficient parallelization. Monitoring GPU utilization helps in diagnosing performance issues and in evaluating whether hardware resources are being fully utilized or underused.

- **How it's done:** Measured using tools like `nvidia-smi`, GPU utilization is sampled continuously or periodically during the inference phase. Either the average utilization across the entire inference run or the peak utilization observed is recorded as the final metric. This helps in understanding the computational load and identifying underutilization or potential inefficiencies in the inference pipeline.

$$GPU = \left(\frac{Tactive}{T_{total}}\right) \times 100\%$$

**Where:**

- $T_{active}$: Time GPU was active under load
- $T_{total}$: Total inference duration

A higher utilization indicates efficient GPU use, while low values may point to bottlenecks (e.g., I/O or CPU-bound operations).

### 4.3.3 Temperature:

Reports the GPU temperature in degrees Celsius, which is a key indicator for detecting thermal stress during inference. Excessive heat can lead to thermal throttling, where the GPU deliberately slows down to prevent damage, resulting in reduced performance. Monitoring temperature helps ensure that the GPU is operating within a safe thermal range and can also indicate if cooling systems (like fans or heatsinks) are functioning properly.

- **How it's done:** Real-time temperature is monitored using system tools (e.g., `nvidia-smi`), and either the peak or average temperature during inference is recorded. This provides a reliable indication of thermal load and helps in evaluating the effectiveness of cooling solutions under different computational workloads.

$$GPU = \max(T_t) \quad for \quad t \in [tstart, t_{end}]$$

**Where:**

- $T_t$: Temperature at time $t$
- Measured in °C

Optimal temperatures typically range between **60°C** to **85°C** depending on the GPU. Exceeding thermal limits may throttle performance.

#### 4.3.4 Power Draw:

Power Draw measures the amount of electrical power (in watts) consumed by the GPU during the inference process. It reflects the energy cost associated with running the model and serves as an indicator of the computational load and efficiency. Higher power draw typically means the GPU is being pushed closer to its maximum capacity, while lower values might indicate underutilization or more optimized execution. Tracking this metric is especially important for evaluating energy efficiency, system stability, and deployment feasibility—particularly in power-constrained or large-scale settings.

- **How it's done:** Live GPU power draw is sampled during inference using tools like `nvidia-smi`, and peak or average power usage is recorded. This provides a quantitative measure of energy consumption during model execution and helps in comparing the efficiency of different model variants or optimization strategies.

$$Power = \frac{1}{T} \int t = 0^T P(t)\, dt$$

**Where:**

- $P(t)$: Power drawn at time $t$ (in Watts)
- $T$: Total inference duration

**Lower power draw** for the same output quality means better energy efficiency — important for scaling and sustainable deployment.

We also plotted a VRAM vs. Time graph to visualize memory usage trends throughout the entire video generation timeline. This graph captures how GPU memory (VRAM) consumption evolves from the start of the process to the end, including key phases such as model loading, latent initialization, denoising steps, and frame decoding. By observing the fluctuations in memory usage over time, we can better understand which stages are most memory-intensive and identify opportunities for optimization or early detection of potential memory bottlenecks.

## 4.4 Other Metrics:

### 4.4.1 Load Time:

Measures the duration required to load the WAN 1.3B model and its associated weights into GPU memory before inference begins. This includes the time taken to allocate memory, initialize model layers, and transfer the weights from disk (or RAM) to the GPU. Load Time is an important metric in evaluating startup latency, especially for deployment scenarios where models are frequently loaded and unloaded (e.g., on-demand or serverless inference).**How it's done:** Load Time is calculated by recording timestamps at the beginning and end of the model initialization process. The difference between these two points gives the total loading duration. This helps assess the overhead introduced before actual inference and is useful for identifying delays due to I/O, disk speed, or inefficient memory allocation.

$$Load = tready - t_{load\_start}$$

**Where:**

- $t_{load\_start}$: Time when model loading begins
- $t_{ready}$: Time when the model is ready for inference
- Measured in **seconds**

**A lower load time** enables faster startup and responsiveness in real-time or on-demand systems.

## 5 Applying optimization techniques on the base model:

- Implemented optimization methods such as LoRA, Operator Fusion, and Quantization on the base WAN 1.3B model to enhance performance and efficiency during video generation.

### 5.1 LoRA

LoRA, short for *Low-Rank Adaptation*, is a method developed to efficiently fine-tune large models by introducing trainable low-rank matrices into existing frozen weights, instead of updating all model parameters. This is especially useful in large-scale models like WAN-1.3B, where updating every parameter is computationally expensive. LoRA addresses this inefficiency by freezing the original weight matrices, training only a small set of low-rank matrices, and then combining these with the original weights at inference time. This leads to significantly reduced training costs while maintaining model performance.

To understand the intuition behind LoRA, consider the task of fine-tuning a video diffusion model. Normally, this would require modifying billions of parameters, which is akin to repainting an entire wall to fix a small scratch. LoRA takes a smarter approach: instead of repainting, it simply adds a small patch where needed. The inefficiencies of full fine-tuning include high GPU memory usage, slow training times, and lack of modularity. LoRA overcomes these by approximating the full weight update $\Delta W$ using two smaller matrices $A$ and $B$, such that the updated weight becomes $W' = W + \Delta W = W + (A \times B)$.

Mathematically, if $W$ is a weight matrix in a transformer's attention mechanism (such as $W_q$, $W_k$, or $W_v$), LoRA keeps $W$ frozen and introduces trainable matrices $A \in R^{d_{out} \times r}$ and $B \in R^{r \times d_{in}}$, where $r$ is a small rank (e.g., 4 or 8). The product $A \times B$ approximates the full update with significantly fewer parameters. For example, if $W$ is a $1024 \times 1024$ matrix (over 1 million parameters), and $r = 4$, then $A$ and $B$ together only have about 8,000 parameters—yielding over 99% reduction in trainable parameters.

The term "low-rank" refers to this approximation strategy where the update matrix $\Delta W$ is constrained to be of low rank, allowing it to capture essential changes without the overhead of full-rank updates. This not only preserves the original model's knowledge (since $W$ is frozen), but also allows for efficient and scalable fine-tuning.

In practice, LoRA retains nearly the same video generation quality as full fine-tuning when configured correctly. The rank $r$ plays a critical role: smaller $r$ values lead to faster training and lighter models with slightly reduced accuracy, while higher $r$ values come closer to full fine-tuning quality at a greater computational cost. LoRA works especially well in video diffusion tasks, maintaining strong temporal consistency and often improving CLIP Score (semantic alignment). However, if not tuned carefully, it might slightly degrade perceptual similarity metrics like LPIPS.

The benefits of LoRA are substantial. Since only 0.1–1% of model parameters are trained, it becomes feasible to fine-tune large models even on consumer-grade GPUs. Moreover, LoRA modules are modular and lightweight—each file is only a few MBs and can be easily swapped to apply domain-specific behaviors (e.g., dancing, nature, anime). This plug-and-play characteristic makes LoRA highly flexible and practical for rapid iteration and deployment in resource-constrained environments.

#### 5.1.1 How We Implemented LoRA

In our project, we integrated LoRA into the video generation pipeline using a modular function-based approach. The implementation was handled through a custom helper function `apply_LoRA()`, which dynamically loads and fuses LoRA weights into the pre-trained WAN-1.3B model at runtime. These LoRA weights were pre-trained and stored locally in the directory `./Wan2.1-T2V-1.3B-crush-smol-v0/`.

The function begins by checking whether LoRA has already been applied using a global flag `is_LoRA_applied`, which prevents redundant re-application. It then verifies the presence of the LoRA weight directory and loads the adapter weights using `pipe.load_lora_weights()`. Once the weights are successfully loaded, LoRA is enabled via `pipe.enable_lora()`, allowing the model to incorporate the low-rank matrices into its inference pipeline.

To further enhance runtime efficiency, we attempt to fuse the LoRA weights directly into the model using `pipe.fuse_lora()`, which effectively merges the low-rank updates with the base model, reducing computation overhead during generation. This step is particularly useful when deploying the model in real-time or low-latency applications.

All operations are wrapped in `try-except` blocks for robust error handling. If any step fails, appropriate error messages are logged, ensuring that the rest of the pipeline continues to function smoothly. This flexible and safe implementation enabled us to fine-tune and adapt the base model to new styles and domains while keeping the memory footprint low and performance high.

#### 5.1.2 Improvements with LoRA:

- **CLIP Score:** Improved alignment between prompt and video content.
- **LPIPS Score:** Temporal consistency is preserved across frames.
- **Latency & Throughput:** Negligible impact on inference speed and frame rate.

## 5.2 4-bit Quantization

4-bit quantization is a model compression technique that reduces the precision of neural network weights and activations from the standard 16-bit or 32-bit floating-point format to just 4 bits. This significantly decreases memory usage and improves inference speed, making it especially effective for large-scale models like WAN-1.3B. When models are too large to fit into GPU memory or need to be run efficiently on edge devices, 4-bit quantization helps by compressing the model size, lowering memory bandwidth requirements, and accelerating matrix operations. In this approach, each weight is represented by only 4 bits, allowing for just 16 distinct values and reducing memory footprint by up to 8× compared to FP32.

To build intuition, imagine a neural network as a symphony recorded in high-fidelity 32-bit audio. While it sounds excellent, the file size is massive. Quantization is like converting that recording into a compressed 4-bit MP3 — it still sounds good to most ears, but is far more efficient to store and transmit. Similarly, quantization trades off a small amount of numerical precision for substantial gains in speed and memory efficiency.

Technically, the process begins by clipping the range of all weights (e.g., from $-1$ to $+1$), then mapping this range into 16 uniform bins. Each weight is replaced by the nearest bin center and only the index of the bin is stored. During inference, these bin indices are dequantized back to approximate floating-point values using a simple lookup table. Mathematically, the quantized weight $w_{q4}$ is computed as:

$$w_{q4} = scale \times q + zero\_point$$

where $q$ is a 4-bit integer between 0 and 15, and `scale` and `zero_point` are learned or calibrated parameters. Techniques like GPTQ, AWQ, or bitsandbytes often optimize these values on a per-channel or per-layer basis to minimize any loss in accuracy.

The benefits of 4-bit quantization are substantial. Compared to full-precision (FP32) models, it can reduce model size to around 12.5% of the original, lower memory usage significantly, and improve inference speed by 2× to 4×. It also enables models to run on more affordable and accessible GPUs like NVIDIA T4 or L4, rather than relying on high-end A100 or H100 hardware.

For video generation models like WAN-1.3B, 4-bit quantization greatly enhances deployability on resource-constrained systems, enabling larger batch sizes and better parallelism. While there may be a minor loss in precision—such as a slight drop in LPIPS (perceptual quality)—semantic alignment measured via CLIP score often remains stable or may even improve. Additionally, modules like LayerNorm, Attention, and FFN are quantized carefully to avoid numerical instabilities. Importantly, the model does not need to be retrained; post-training quantization (PTQ) usually suffices. Quantization can also be combined with other techniques like LoRA and Operator Fusion for cumulative performance benefits. Modern libraries like `bitsandbytes`, `GPTQ`, and `AutoAWQ` support 4-bit quantization with minimal additional effort, offering a robust path to fast, memory-efficient, and scalable inference.

#### 5.2.1 Improvements from Quantization (4-bit):

- **Warm-up Time:** Reduced significantly as quantized weights are smaller and load faster.
- **Throughput:** Increased frame generation speed due to efficient 4-bit computations.
- **GPU Memory Usage:** Lower memory consumption because of reduced weight size , Peak VRAM usage decreased.
- **Latency:** Significant improvement in both Frame-wise and Clip-wise Latency.

### 5.3 Operator Fusion

**What is Operator Fusion?** Operator Fusion is an optimization technique where multiple small operations (like `add`, `matmul`, `ReLU`, etc.) are merged or "fused" into a single, larger GPU operation. This significantly reduces the overhead time spent launching and switching between GPU kernels. Instead of executing each operation individually, Operator Fusion combines them into a single fused kernel, improving both speed and computational efficiency.

**Beginner Intuition:** Imagine cooking a three-course meal. If you call the chef separately for each dish, a lot of time is wasted as they repeatedly enter, prepare, cook, and exit the kitchen. Operator Fusion is like calling the chef once to cook everything in one go—less back-and-forth means faster results. In machine learning terms, each tensor operation incurs GPU overhead; fusing them reduces this overhead, resulting in faster training and inference.

**How It Works (Technically):** We used `torch.compile()` from PyTorch 2.0, which analyzes the computation graph, detects chains of operations like `add`, `ReLU`, `matmul`, etc., and fuses them into a single optimized kernel. This fusion occurs just-in-time (JIT) during the first forward pass, so there is no need to rewrite the model. PyTorch handles the optimization automatically.

**What Changes Did We Make?** The only modification was enabling `torch.compile()`—no architectural changes were needed. PyTorch internally performs all fusion and optimization.

**Performance Impact:** During the first generation, there might be a slight delay due to compilation and warm-up. However, it is still faster than the baseline. From the second generation onward, throughput improves significantly, and latency drops noticeably.

**Does It Affect Video Quality?** No. There is no degradation in LPIPS (perceptual quality) or CLIP Score (semantic alignment). Operator Fusion only speeds up inference without compromising output quality.

**Benefits Summary:** Operator Fusion provides faster inference without affecting video quality. It does not require retraining, works alongside techniques like quantization and LoRA, and is completely automatic via `torch.compile()`. It is a safe and effective way to accelerate deep learning models.

#### 5.3.1 Improvements from Operator Fusion:

- **Execution Efficiency:** Merges multiple GPU kernel operations into a single fused kernel, reducing overhead from multiple kernel launches.
- **Reduced Latency in Batch Mode:** When prompts are processed in batches, fusion time is distributed, leading to a net decrease in inference time and an increase in throughput.
- **Consistent Output Quality:** Maintains CLIP and LPIPS scores close to baseline values, ensuring no significant drop in generated video quality.

#### 5.3.2 Final Improvements:

The above mentioned optimization techniques boosted the performance of model. For comparision, we generated a sample video - 480p, 1 frame, 1 fps, "A cat walking on the moon" - using the baseline and optimized pipeline with each optimization technique individually as well as combined. We also ran it locally by cloning the repo and enabling the flash attention. These are the results:

### 5.4 Why Attention Needs Optimization

Self-attention operations have a memory complexity of O(n 2 ), where

n is the number of input tokens or spatial locations. This is because the attention mechanism computes pairwise relationships between all inputs simultaneously.

In large models like WAN 1.3B, especially when generating images or videos, this leads to very large attention maps that can exceed the memory capacity of typical GPUs. Without optimization, the model may run into out-of-memory errors during inference.

Attention slicing helps alleviate this issue by reducing the size of the attention computation at any given time, making large-scale models more memory-efficient and easier to deploy in constrained environments.

#### 5.4.1 Attention Slicing

Attention slicing is a memory optimization technique used to reduce peak GPU memory usage during inference in models with self-attention layers, such as transformers or U-Nets in diffusion models. Instead of computing the full attention operation over all input tokens at once—which creates large intermediate tensors—attention slicing divides the query matrix into smaller slices and processes each slice sequentially.

This allows the model to reuse memory across slices, significantly lowering the peak memory footprint. While this may slightly slow down inference due to repeated computation, it enables large models to run on GPUs with limited VRAM without sacrificing output quality.

#### 5.4.2 Implementation of Attention Slicing

To reduce memory usage during inference, we enabled attention slicing in the WAN 1.3B model using Hugging Face's `diffusers` library. This was achieved by calling the following method after loading and moving the pipeline to the GPU:

```
pipe.enable_attention_slicing()
```

This command modifies the attention computation to process smaller slices of the input sequentially, instead of computing the full attention map at once. As a result, the peak GPU memory consumption is significantly reduced. This allowed us to run inference reliably on GPUs with limited VRAM (e.g., 12 GB) without modifying the model architecture.

For even finer control, a specific slice size can be set:

```
pipe.enable_attention_slicing(slice_size=1)
```

**Default Behavior (slice_size="auto")**

The library automatically chooses a slice size based on the model architecture and available GPU memory.

Internally, it typically sets the slice size to half the attention head dimension. For example, if the attention head has dimension 64, the default slice size might be 32. This default strikes a balance between:Reducing memory usage, and Minimizing slowdown in inference.

### 5.4.3 Improvements from Attention Slicing:

- **Load Time:** It has increased the Load time as compared to the base model.

## 5.5 CPU Offloading

**What is CPU Offloading?** CPU Offloading is an optimization technique where some parts of the model computation—typically memory-intensive or less parallel operations—are shifted from the GPU to the CPU. This reduces GPU memory usage, making it possible to run larger models or higher-resolution prompts without encountering out-of-memory (OOM) errors. While CPUs are slower than GPUs, they are capable of handling certain operations efficiently, enabling smoother inference and greater compatibility with limited hardware.

**Beginner Intuition:** Imagine trying to move heavy furniture using a single elevator (GPU). If the elevator is overloaded, it gets stuck. Instead, sending a few items up the stairs (CPU) helps the process continue smoothly. Similarly, CPU Offloading reduces the load on the GPU, allowing the entire system to function without memory crashes, even if a bit slower.

**How It Works (Technically):** We enabled `device_map="auto"` using Hugging Face's libraries like `diffusers` and `transformers`. This splits the model across the CPU and GPU depending on available memory. Layers that cannot fit into GPU memory are automatically offloaded to the CPU. PyTorch manages data transfer between devices during runtime, requiring no manual intervention.

**What Changes Did We Make?** The only modification was enabling CPU Offloading via `device_map`—no architectural changes were needed. The memory management and offloading logic are handled internally by the Accelerate library and PyTorch.

**Performance Impact:** During the first generation, there might be a slight increase in latency due to data transfer between GPU and CPU. However, it prevents crashes and enables the use of larger models on limited VRAM setups. For most use cases, performance remains stable and inference times are acceptable.

**Does It Affect Video Quality?** No. There is no degradation in LPIPS (perceptual quality) or CLIP Score (semantic alignment). CPU Offloading does not alter model outputs and only impacts the memory distribution strategy.

**Benefits Summary:** CPU Offloading improves memory efficiency and allows deep learning models to run on devices with low GPU memory. It works without retraining, requires no code changes, and integrates seamlessly with other optimization techniques like Operator Fusion, Flash Attention, and Quantization.

### 5.5.1 Final Improvements:

The above mentioned optimization technique boosted the performance of model under memory constraints. For comparison, we generated a sample video - 480p, 1 frame, 1 fps, "A cat walking on the moon" - using the baseline and optimized pipeline with each optimization technique individually as well as combined. We also ran it locally by cloning the repo and enabling the flash attention. These are the results:

### 5.5.2 Improvements from CPU Offloading:

- **GPU Memory Savings:** Significant reduction in Max VRAM usage by offloading model weights and intermediate tensors to CPU memory.
- **LPIPS:** Significant increase in LPIPS score from 0.138 to 0.285.
- **Stable Execution:** Reduces the risk of out-of-memory (OOM) errors during inference by distributing memory load.

## 5.6 Classifier-Free Guidance (CFG) and CFG Parallelism

Classifier-Free Guidance (CFG) is the core mechanism that enables high-fidelity, prompt-adherent image and video generation. It works by steering the generation process at each denoising step through the combination of two separate predictions.

### 5.6.1 Standard Sequential CFG Process

The standard process for a single denoising step is as follows:

1. **Generate Positive Prediction:** The model's U-Net performs a forward pass using the embeddings from the positive prompt (*what you want to see*). This produces a `positive_prediction`.
2. **Generate Negative Prediction:** The U-Net performs a second, separate forward pass using the embeddings from the negative prompt (*what you want to avoid*). If no negative prompt is given, this pass uses empty text embeddings. This produces a `negative_prediction`.
3. **Combine Predictions:** The two predictions are combined using the CFG formula, where the `guidance_scale` determines the strength of the effect:

$$Final\_Prediction = negative\_prediction + guidance\_scale \cdot (positive\_prediction - negative\_prediction)$$

This `Final_Prediction` is then used to denoise the image for the current step. The critical bottleneck here is the two separate forward passes through the resource-intensive U-Net for every single step. For a 50-step inference, this results in 100 total U-Net passes.

### 5.6.2 Introducing CFG Parallelism: The Optimization

CFG Parallelism is a performance optimization technique that eliminates the need for two sequential forward passes. Instead of processing the positive and negative prompts separately, it batches them together to be processed simultaneously.

The parallelized process for a single denoising step is:

1. **Batch Prompts:** The text embeddings for the positive and negative prompts are concatenated into a single, larger tensor. For example, if each prompt embedding has shape $[1, 77, 1024]$, they are stacked to create a tensor of shape $[2, 77, 1024]$.
2. **Perform a Single Forward Pass:** This combined batch is fed into the U-Net in one forward pass. Since GPUs are optimized for parallel processing, performing one pass on a batch of size 2 is significantly faster than two separate passes of size 1.
3. **Split Predictions:** The output from the U-Net is split back into `positive_prediction` and `negative_prediction`.
4. **Combine Predictions:** These are plugged into the same CFG formula:

$$Final\_Prediction = negative\_prediction + guidance\_scale \cdot (positive\_prediction - negative\_prediction)$$

The result is mathematically identical to the sequential method, but the time taken is drastically reduced—often achieving a $\sim 1.8\times$ speedup because the overhead of launching two U-Net computations per step is eliminated.

### 5.6.3 CFG Parallelism in the Wan 1.3B Model

In the standard diffusers pipeline (as seen in `Wan-AI/Wan2.1-T2V-1.3B-Diffusers`), the sequential CFG process is used by default. When you provide a negative prompt:

- **Guidance Activated:** If `guidance_scale` $> 1.0$, CFG is active.

– **Negative Prompt Utilized:** The pipeline uses the negative prompt embeddings for the unconditional part of the CFG calculation.
– **Sequential Execution:** The U-Net first runs with positive prompt embeddings, then with negative prompt embeddings, before combining results.

Thus, In the Hugging Face diffusers library, Classifier-Free Guidance (CFG) and its performance optimization, CFG Parallelism, are intrinsically linked and enabled by default. Whenever an inference is run with a `guidance_scale` greater than 1.0, the pipeline automatically performs CFG Parallelism. It achieves this by batching the conditional embeddings (from the positive prompt) and the unconditional embeddings (from the negative prompt or an empty string) into a single tensor

#### 5.6.4 Sources Confirming the Link Between CFG and Negative Prompts

Several sources, ranging from original research papers to authoritative technical explanations, confirm the link between Classifier-Free Guidance (CFG) and the concept of negative prompts.

**Primary Source: The Original Research Paper** The foundational concept originates from the paper that introduced the technique. While it does not explicitly use the term "negative prompt" (which was popularized later by the community), it describes the exact mechanism.

– **Classifier-Free Diffusion Guidance (Ho & Salimans, 2022)** https://arxiv.org/abs/2207.12598 *What to look for:* See Section 3, specifically Equation (5). The paper describes a guidance technique that mixes the score estimate of a conditional model (guided by a class label $c$) and an unconditional model. The unconditional part is now replaced by the negative prompt in modern usage. The guidance scale $w$ in the paper corresponds to the CFG scale. This paper is the origin of the formula:

$$Final = Unconditional + Scale \times (Conditional - Unconditional)$$

which shows that guidance works by steering away from the unconditional prediction and towards the conditional one.

**Authoritative Technical Explanations (Hugging Face)** Hugging Face provides widely trusted documentation that explains CFG in practical terms:

– **Hugging Face Diffusers Documentation:** `StableDiffusionPipeline`(uses the generic DiffusionPipeline class to load and run the specific Stable Diffusion model) https://huggingface.co/docs/diffusers/v0.10.0/en/api/pipelines/stable_diffusion#diffusers.StableDiffusionPipeline.__call__ *What to look for:* The `__call__` method parameters include `prompt` (positive prompt), `negative_prompt`, and `guidance_scale` (CFG scale). The documentation explicitly states that `guidance_scale` guides generation away from the negative prompt embeddings, showing how the CFG formula is implemented in code.

These sources collectively provide a comprehensive view—covering the original academic theory, practical implementation in code, and intuitive community explanations—confirming that using a negative prompt is an integral part of the Classifier-Free Guidance mechanism.

## 6 The Role Of Diffusion Schedulers

In a diffusion model pipeline, the Scheduler is a critical component that dictates the entire denoising process. The U-Net model predicts the noise (or a related quantity) present in the image at a given timestep, but it is the scheduler's responsibility to use that prediction to compute the less noisy image for the next step.

The scheduler's algorithm determines:

- The sequence and values of timesteps to denoise at.
- How to combine the model's prediction with the current noisy image.
- The trade-off between inference speed (number of steps) and output quality.

The diffusers library provides a wide variety of schedulers, each with different characteristics. However, schedulers are not universally interchangeable due to a crucial factor: the model's prediction type.

### 6.1 The Critical Factor: `prediction_type`

A diffusion model's U-Net can be trained to predict one of several targets. The choice of target is a fundamental architectural decision made during training. The three main types are:

1. **Epsilon (Noise Prediction)**: The most common and traditional approach, used in models like Stable Diffusion v1.5. The model is trained to predict the exact noise that was added to the image. The scheduler's role is to subtract this predicted noise from the noisy input.
2. **v_prediction (Velocity Prediction)**: A more modern approach used in models like Stable Diffusion 2.1 and SDXL. The model predicts a "velocity" vector representing the direction of the denoising process. This can lead to improved generation stability and better quality at lower step counts.
3. **flow_prediction (Flow Prediction)**: Used in newer paradigms such as "flow matching" (e.g., Stable Cascade). Instead of predicting noise or velocity, the model predicts a vector field mapping a noisy image directly to a less noisy one. Schedulers for these models are incompatible with epsilon or v_prediction models.

A mismatch between the model's trained `prediction_type` and the scheduler's expected type will cause runtime errors—e.g.,unsupported epsilon or `flow_prediction` errors.

### 6.2 Analysis of the Wan2.1-T2V-1.3B Model and Scheduler Compatibility

The successful integration of the `DPMSolverMultistepScheduler` (DPM) is directly attributable to its modern, flexible design and the Wan 1.3B model's specific architecture:

- **Model Architecture:** The `Wan-AI/Wan2.1-T2V-1.3B-Diffusers` model uses `prediction_type: 'v_prediction'`. It does not output epsilon, explaining why schedulers expecting strictly epsilon predictions will fail.
- **Why DPM Worked:** Modern schedulers, including DPM, are designed to be *model-aware*. During initialization via `.from_config()` or `.from_pretrained()`, they read the model's `scheduler_config.json` to automatically detect the `prediction_type` and adjust their internal algorithms accordingly.
- This automatic adaptation is why DPM works seamlessly with Wan 1.3B without errors.

### 6.3 Feature Comparison: Common Schedulers

While many schedulers exist, some traditional ones can cause issues when paired with `v_prediction` models like Wan 1.3B:

- **LMSDiscreteScheduler (LMS):** Often fast and good for high-quality still images, but it assumes epsilon prediction. It will fail or produce degraded outputs with `v_prediction` models unless modified.
- **DDIMScheduler (DDIM):** Popular for its balance of quality and speed. However, its default implementation also assumes epsilon prediction, leading to incompatibility errors.

  - **Euler / Euler Ancestral Schedulers:** Known for sharp, detailed generations but tuned for epsilon prediction. Using them with a `v_prediction` model can cause artifacts or runtime mismatches.

In summary, choosing a scheduler requires checking both the model's `prediction_type` and the scheduler's supported modes. For Wan 1.3B, DPM provides a safe, performant choice, while LMS, DDIM, and Euler require architectural alignment or modifications.

## 7 Gradio Interface for Model Monitoring

The app is built using Gradio's `Blocks` API and allows users to generate synthetic videos based on text prompts and optimization settings, while monitoring hardware usage and generation metrics. The user interface is visually structured, modular, and interactive.

### 7.1 Theme Customization

The UI employs a custom theme based on `gr.themes.Base`:

- **Primary Hue:** Indigo, which defines the dominant accent color used in primary interactive components such as sliders, buttons, and highlights. It reinforces the visual identity, provides a consistent look, and draws attention to key actions.
- **Secondary Hue:** Gray, applied to backgrounds, containers, and secondary elements. This offers a neutral visual base, improving content focus while ensuring a clean, professional appearance through balanced contrast.
- **Radius Size:** Soft corner radii ranging from 6 px to 16 px, applied proportionally to different UI components. This produces a modern, approachable interface while maintaining consistency across elements.
- **Spacing Size:** Carefully tuned padding and margin values between 2 px and 32 px, ensuring readable layouts, comfortable white space, and avoidance of visual clutter.

Additionally, the `.set()` method is employed to inject fine-grained, CSS-like customizations, including:

- **Backgrounds:** A diagonal gradient for the body background and semi-transparent overlays for content blocks, creating a layered glassmorphism effect.
- **Text and Color Scheme:** High-contrast white body text on dark backgrounds for readability, and consistent accent colors for interactive elements.
- **Borders and Shadows:** Subtle semi-transparent borders combined with deep drop shadows to add depth and separation between interface layers.
- **Input Fields:** Transparent backgrounds with matching borders to maintain harmony with container styles.
- **Buttons:** Indigo-based background colors with hover-state intensity changes, paired with white text for clarity.

Custom CSS rules further refine the design by introducing:

- Modern typography using the Inter font family for a clean, contemporary appearance.
- Glassmorphism effects via backdrop blur, 3D perspective transforms, and soft transparency.
- Subtle entry animations such as `fadeIn` for smooth UI load transitions.
- Enhanced sliders, accordions, and buttons with distinctive hover and active states to improve interactivity cues.

This combination of Gradio's theming API and CSS overrides achieves a balance between consistency and creative freedom, enabling a refined, visually appealing interface without leaving the Python environment.

## 7.2 CSS Reset and Animation Effects

A CSS reset is injected using `gr.HTML()` to:

- Load the `Inter` font for clean, modern typography. This ensures cross-browser consistency and improved aesthetics.
- Add gradient backgrounds and subtle 3D transformations to the interface. These effects help differentiate sections and enhance depth.
- Style accordions, buttons, and sliders with hover/transition effects. It adds responsiveness to user interaction.
- Enable a fade-in animation on load to improve UX aesthetics. Smooth transitions create a polished and professional feel.

## 7.3 UI Structure and Layout

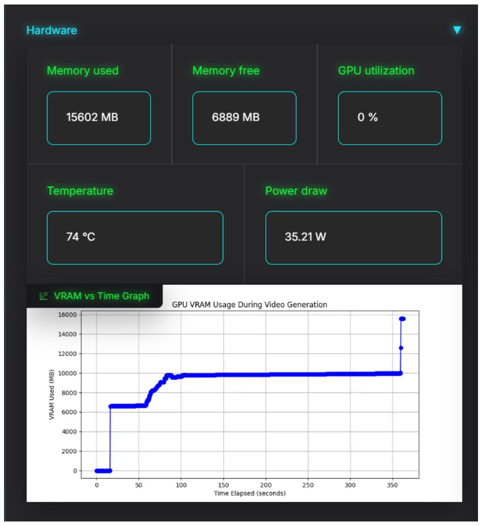
The interface is built using nested Gradio blocks:

- `gr.Blocks`: Root context manager containing all layout elements. It is the container that coordinates all interface components.
- `gr.Group`: Used to cluster related controls, like metrics or settings. Groups improve modularity and help organize complex UIs.
- `gr.Row` and `gr.Column`: Used for horizontal and vertical layout alignment. This allows for responsive design and compact layouts.
- `gr.Accordion`: Collapsible sections to organize metrics and reduce clutter. Accordions prevent information overload and improve navigability.

## 7.4 Metrics Panel

The left section is dedicated to hardware and efficiency monitoring. It contains:

- `Hardware Accordion` Contains five `Textbox` elements for memory, GPU usage, temperature, and power draw. Real-time VRAM usage is plotted with a `Plot` component.



Hardware

Memory used: 15602 MB

Memory free: 6889 MB

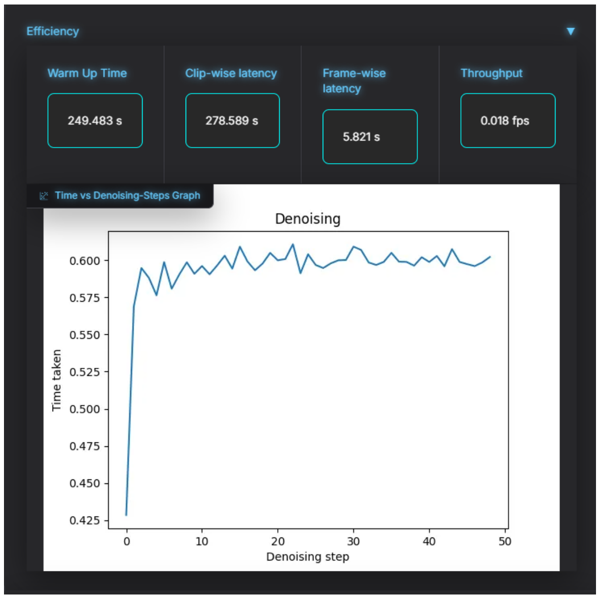
GPU utilization: 0 %

Temperature: 74 °C

Power draw: 35.21 W

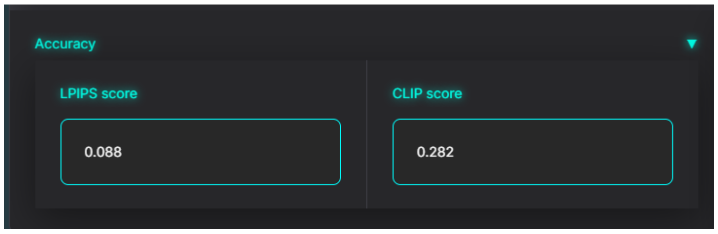
**Hardware Section**

- `Efficiency Accordion:` Displays warm-up time, latency metrics, throughput, and a denoising time graph. These metrics help evaluate performance improvements.



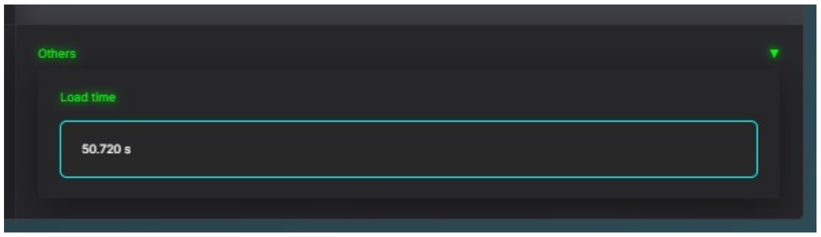
**Efficiency Section**

- `Accuracy Accordion` Shows LPIPS and CLIP scores in read-only `Textbox` elements. These scores give insights into video quality and prompt-image alignment.



Accuracy Section

- `Others Accordion:` Displays load time using Textboxelements. This helps monitor preprocessing and setup duration.



Others Section

## 7.5 Parameter Inputs

This section allows users to configure the video generation:

- `Radio Button`: To choose resolution (240p–720p). Enables users to balance quality and speed based on system capability.
- `Textbox`: One for the generation prompt and one for the negative prompt. These inputs guide the content and avoid unwanted elements.

- `Slider`: For controlling FPS and number of frames. They offer precise control over video length and smoothness.

- `CheckboxGroup`: Lets users select multiple optimization techniques. Provides flexibility in applying model or inference-level improvements.

- `Radio Button`: To select the diffusion scheduler. Scheduler choice can affect both video coherence and generation time.

- `Checkbox`: For reloading the model. This is useful when toggling between optimization modes or changing weights.

## 7.6 Video Output Display

The right-side group displays the generated video:

- A single `Video` component to preview the result. This enables users to immediately view the output without leaving the app.

- The video is displayed with full resolution based on user-selected settings. Users can validate quality and coherence visually.

## 7.7 Generate Button and Logic

The `Generate` button is connected to a function that:

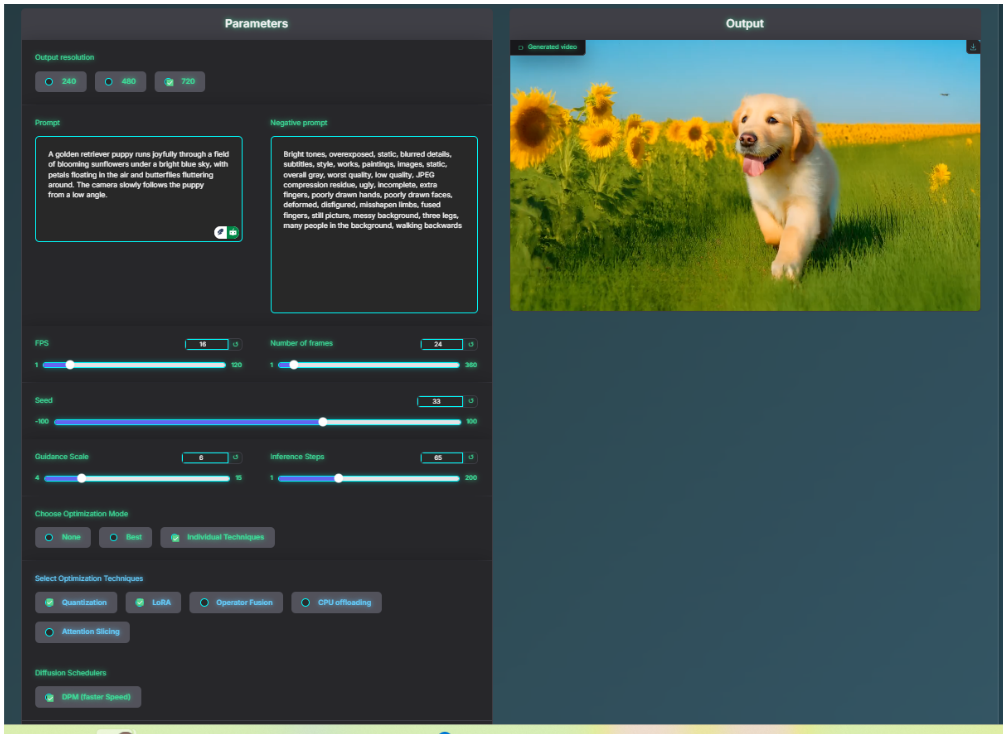
- Starts a separate thread to monitor GPU memory in real-time. This ensures accurate tracking of hardware usage during inference.

- Applies optimizations based on user-selected techniques. These can include quantization, Operator Fusion, Attention Slicing, Lora, Cpu Offloading.

- Generates the video using a backend pipeline and returns multiple outputs:

  - Video path
  - Latency and throughput values
  - LPIPS and CLIP scores
  - VRAM and denoising plots

  These outputs provide holistic per formance insights.



Output resolution: ( ) 240 ( ) 480 [✓ 720]

Prompt: A golden retriever puppy runs joyfully through a field of blooming sunflowers under a bright blue sky, with petals floating in the air and butterflies fluttering around. The camera slowly follows the puppy from a low angle.

Negative prompt: Bright tones, overexposed, static, blurred details, subtitles, style, works, paintings, images, static, overall gray, worst quality, low quality, JPEG compression residue, ugly, incomplete, extra fingers, poorly drawn hands, poorly drawn faces, deformed, disfigured, misshapen limbs, fused fingers, still picture, messy background, three legs, many people in the background, walking backwards

FPS: 16 (1–120)
Number of frames: 24 (1–360)
Seed: 33 (-100–100)
Guidance Scale: 6 (4–15)
Inference Steps: 65 (1–200)

Choose Optimization Mode: ( ) None ( ) Best [✓ Individual Techniques]

Select Optimization Techniques: [✓ Quantization] [✓ LoRA] ( ) Operator Fusion ( ) CPU offloading ( ) Attention Slicing

Diffusion Schedulers: [✓ DPM (faster Speed)]

User Interface

## 7.8 GPU Timer Monitoring

A `Timer` component is used to periodically call an update function:

- Updates metrics like Memory Used, Memory Free, GPU Utilization, Temperature, and Power Draw.
- Uses `threading.Event` to start/stop memory monitoring during the generation window.
- The VRAM usage graph is dynamically updated to reflect hardware behavior over time.
- Ticker and threading make real-time feedback possible without blocking UI or delaying inference.

## 7.9 Summary

The interface is designed for a clean, responsive, and modular UX using Gradio’s most flexible layout tools. By combining a custom theme, real-time performance monitoring, and detailed controls, it provides both usability and depth for video generation tasks. Advanced users can assess system-level behaviors, while new users benefit from a friendly layout and guided input options

# 8 Key Learnings and Contributions

- **Deepened Understanding of the Diffusion Algorithm:** Studied the underlying mathematics and mechanisms that enable the WAN 1.3B model to generate high-quality video sequences.
- **Applied Theory to Improve WAN 1.3B:** Leveraged insights into model design to enhance inference performance and efficiency.
- **Strengthened UI Design via Gradio:** Developed a clean, user-friendly frontend for controlling model inputs and visualizing outputs.
- **Implemented Multiple Optimization Techniques:** Integrated LoRA, quantization, attention slicing, operator fusion, and CPU offloading for improved results.

- **Ensured Compatibility Among Techniques:** Addressed conflicts between optimization layers through profiling, debugging, and iteration.

## 9 Challenges Faced

- **Steep Learning Curve and Multi-Layer Debugging:** Understanding the internal mechanics of WAN 1.3B and integrating multiple optimization techniques required deep technical investigation across PyTorch, Hugging Face, and Gradio layers.
- **LoRA Technique Weights Acquisition:** Sourcing or generating LoRA weights compatible with the WAN 1.3B model was difficult. Available checkpoints were either scarce or incompatible, necessitating manual tuning and experimentation.
- **torch.compile() Compatibility Issues:** Attempts to accelerate execution using `torch.compile()` failed due to unsupported attributes and operations. The feature's experimental nature compounded the problem, forcing us to fall back to more stable methods.
- **Model Loading Challenges in Local Setup:** Loading the WAN 1.3B model locally proved impractical due to large memory requirements, conflicting dependencies, and unclear setup documentation. Eventually, we transitioned to the Hugging Face diffusers pipeline.
- **Inference Strategy Selection:** Selecting the right inference strategy was non-trivial. We had to experiment with batch sizes, execution devices, and multiple optimization combinations, which occasionally conflicted (e.g., quantization with CPU offloading).