# Fast Video Generation

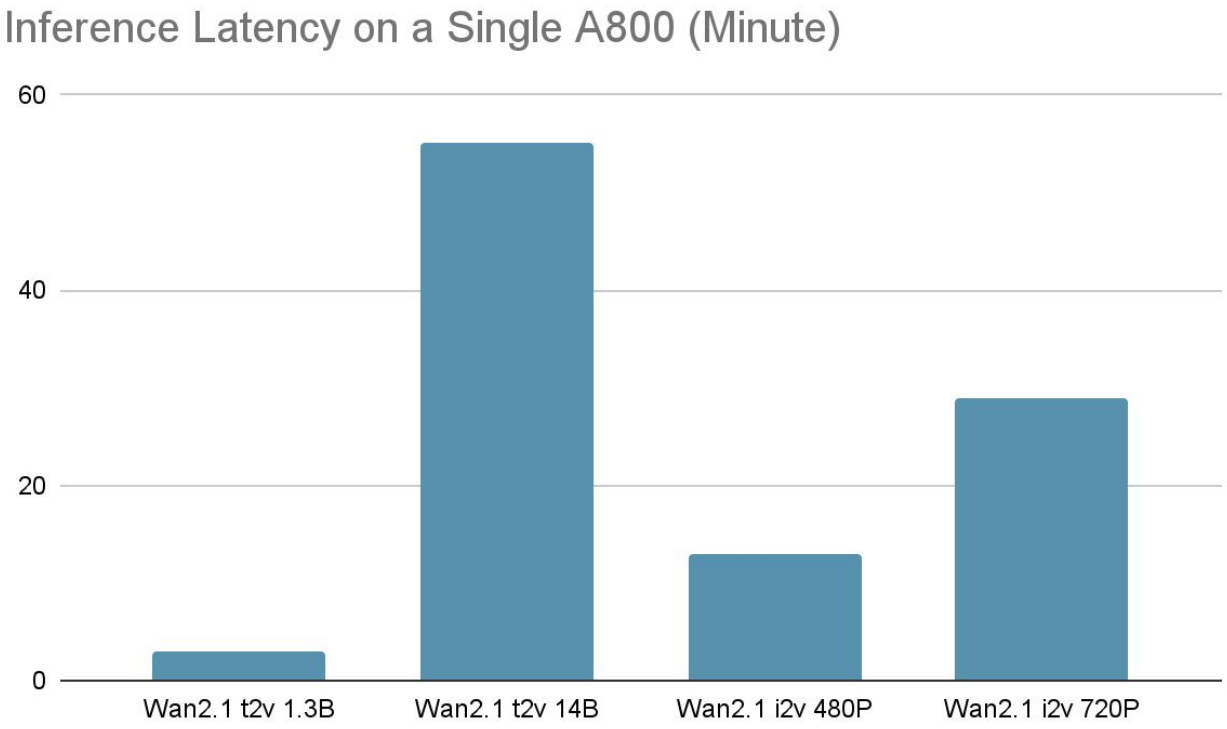
69th Vienna Deep Learning Meetup

Rahim Entezari

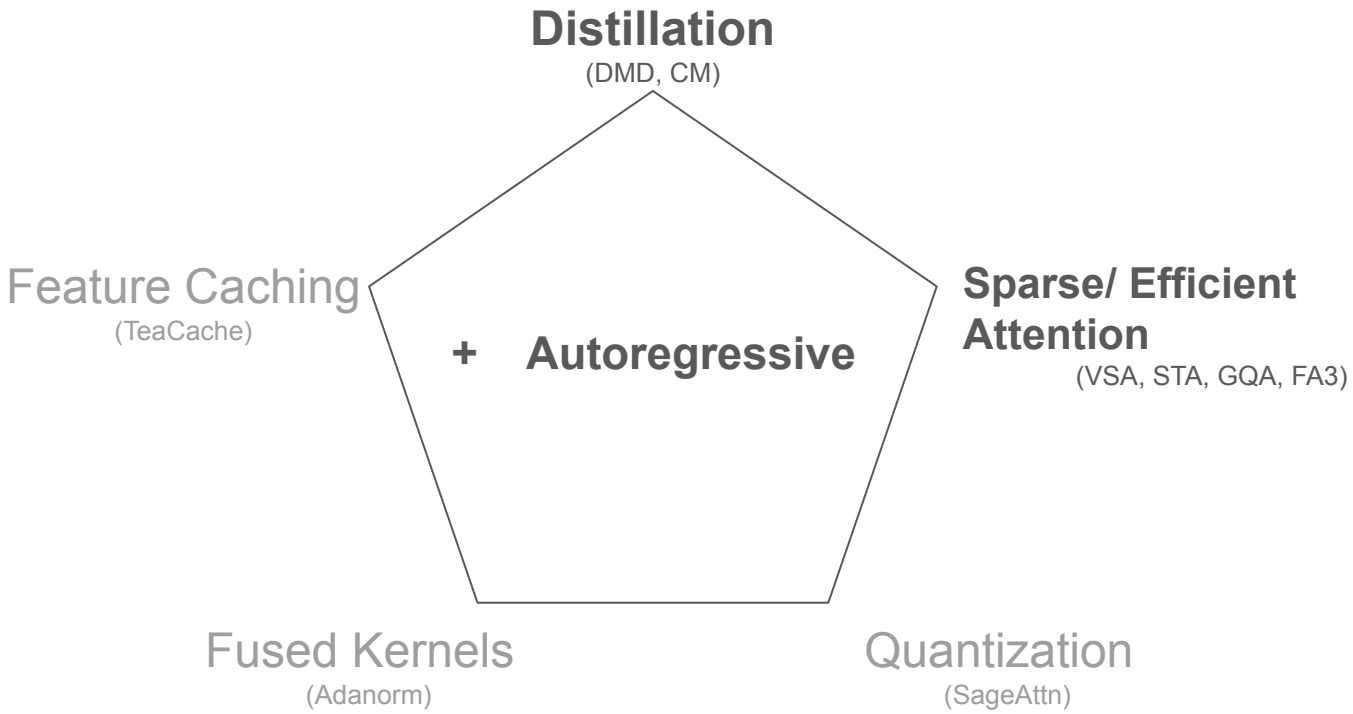
23.10.2025

# Why?

https://nvlabs.github.io/LongLive/#interactive



# Efficiency Dimensions

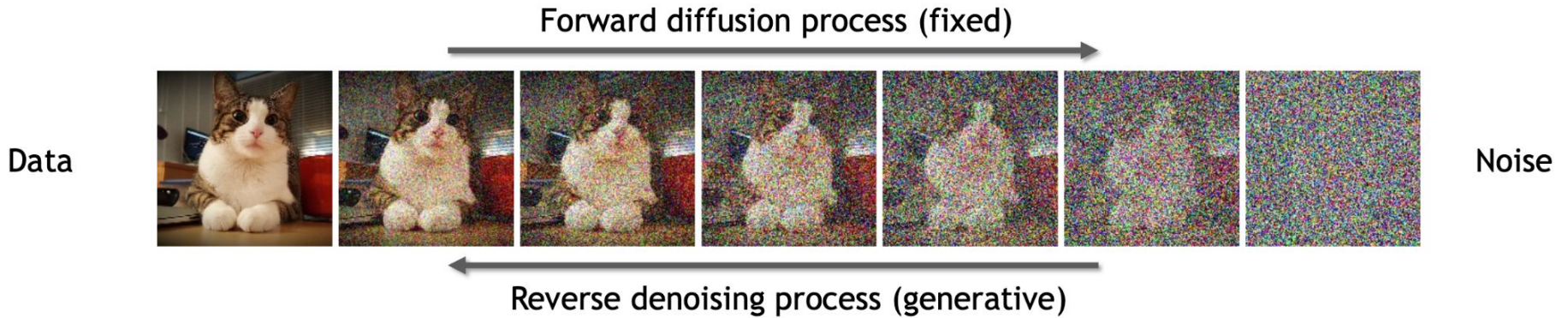


# Standard (video) Diffusion

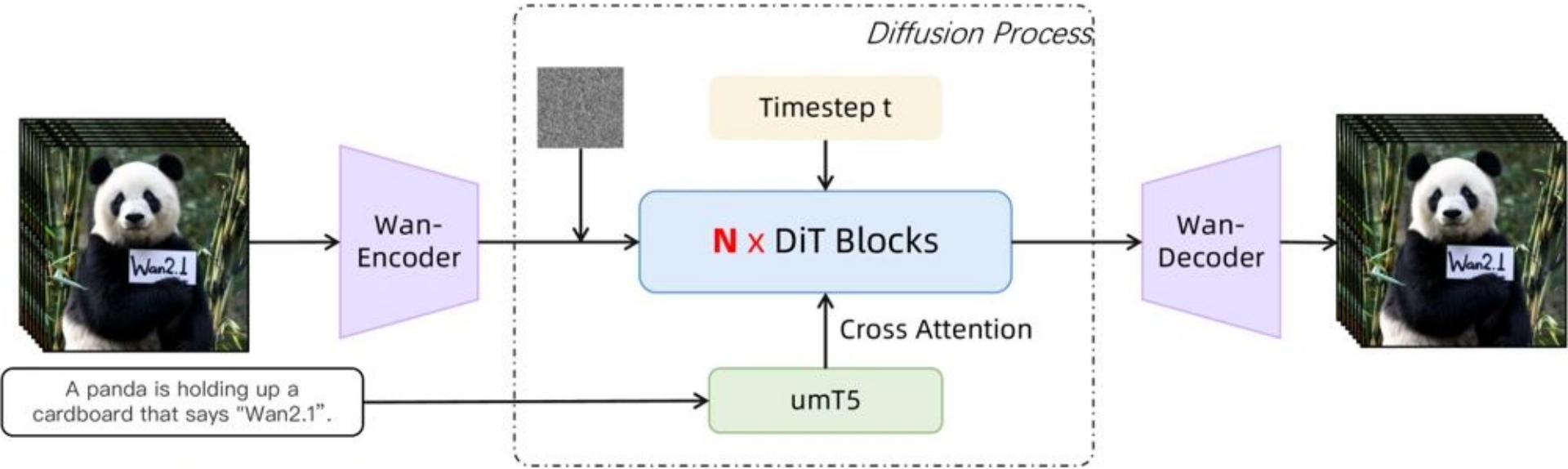
# Diffusion (image)

Denoising diffusion models consist of two processes:

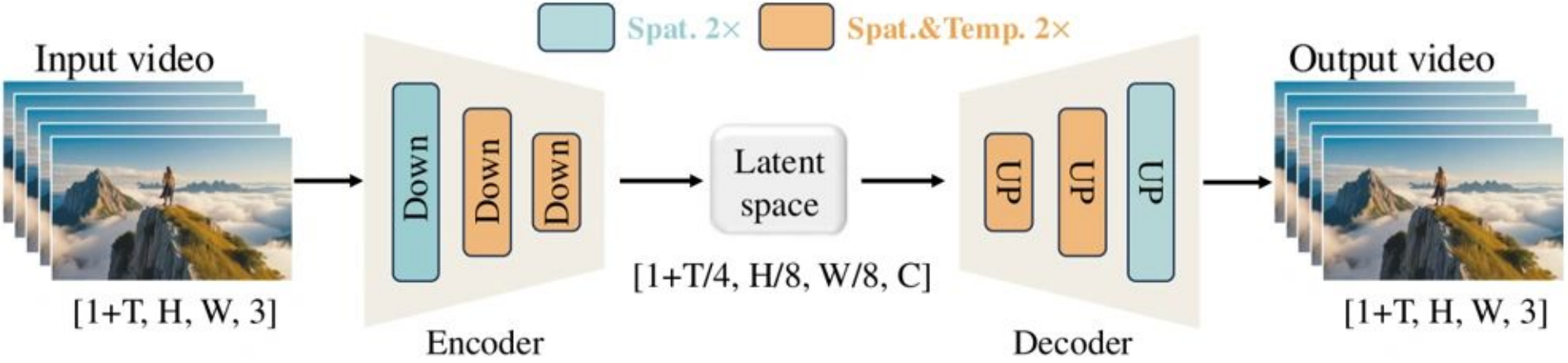
- Forward diffusion process that gradually adds noise to input
- Reverse denoising process that learns to generate data by denoising



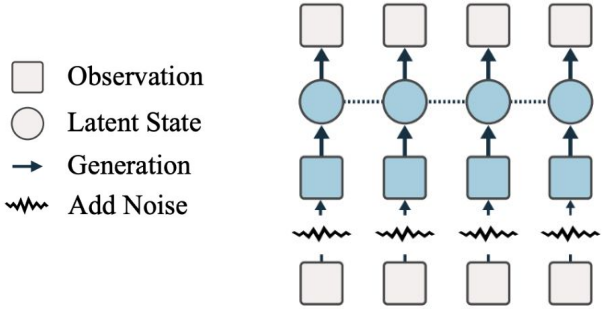
# Architecture: Wan2.1



# VAE


Spat. 2×
Spat.&Temp. 2×
Input video
[1+T, H, W, 3]
Down
Down
Down
Encoder
Latent space
[1+T/4, H/8, W/8, C]
Up
Up
Up
Decoder
Output video
[1+T, H, W, 3]

# Diffusion Video Pipeline




5 sec video (16fps) , with 256px

1. VAE encode (4x8x8 compression) (B, 3, 80, 256, 256)→ (B, 16, 20, 32, 32)
2. Adding noise (timestep *t*)
3. 3D conv (kernel/stride (1, 2, 2)): From (B, 16, 20, 32, 32) to (B, dim=1536, 20, 16, 16)

   $cnn_{output}=\lfloor Stride(Input-Kernel)\rfloor+1$

4. Flatten tokens: (B, 1536, 20, 16, 16)--> (B, S=5120, D=1536)
5. Transformer blocks (denoise) : output: B,S,D
6. Head + unpatchify: Project back to latent patch space, then reshape back to (B, 16, 20, 32, 32)
7. VAE decode: Reconstruct pixels to (B, 3, 80, 256, 256)

# Inside a Transformer block

1. **Pre-Normalization & Modulation**

```
norm_x = self.norm1(x).float()  # (B, 5120, 2048)
modulated_x = norm_x * (1 + e[1]) + e[0]  # Scale + shift
```

**2. QKV Projection**

```
q = self.norm_q(self.q(modulated_x)).view(B, 5120, 16, 128)  # (B, S, H, D)
k = self.norm_k(self.k(modulated_x)).view(B, 5120, 16, 128)  # (B, S, H, D)
v = self.v(modulated_x).view(B, 5120, 16, 128)               # (B, S, H, D)
```

**3. ROPE Application**

```
# 3D rotary encoding for (20 frames, 16×16 spatial patches)
roped_q = rope_apply(q, grid_sizes, freqs, offsets)  # (B, 5120, 16, 128)
roped_k = rope_apply(k, grid_sizes, freqs, offsets)  # (B, 5120, 16, 128)
```

**4. attention**

```
attn_output = attention(
    q=roped_q,            # (B, 5120, 16, 128)
    k=roped_k,            # (B, 5120, 16, 128)
    v=v,                  # (B, 5120, 16, 128)
    k_lens=seq_lens,      # (B,) - actual sequence lengths
    window_size=(-1,-1)
)
# Output: (B, 5120, 16, 128)
```

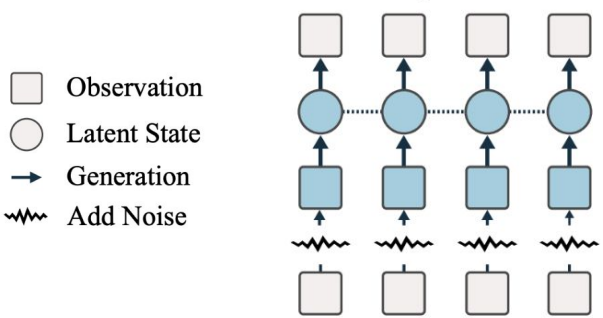
**5. output**

```
attn_output = attn_output.flatten(2)  # (B, 5120, 2048)
attn_output = self.o(attn_output)     # (B, 5120, 2048)
x = x + attn_output * e[2]            # Residual with modulation
```

# Slow video generation: WAN2.1

We generate all frames at once (parallel), with **bi-directional** attention

```
attn_output = attention(
    q=roped_q,            # (B, 5120, 16, 128)
    k=roped_k,            # (B, 5120, 16, 128)
    v=v,                  # (B, 5120, 16, 128)
    k_lens=seq_lens,      # (B,) - actual sequence lengths
    window_size=(-1,-1)
)
# Output: (B, 5120, 16, 128)
```



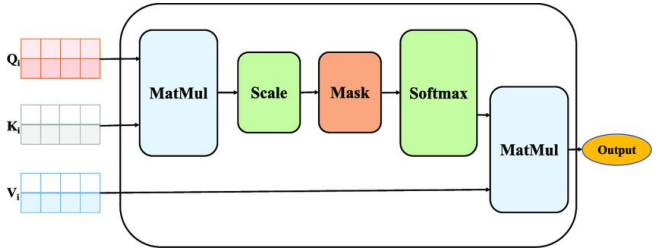
# Autoregressive

# From Diffusion to Autoregression

**Goal:**
To transform a pre-trained video diffusion model into a fast, per-latent-frame causal generator suitable for real-time interactive application.

bidirectional DiT → causal autoregressive architecture (full-attention → **block causal** attention)

# full-attention → block causal attention




Example:

(F,H,W)=(3,2,2), then we flatten these 12 tokens

S = [ f0p0, f0p1, f0p2, f0p3, f1p0, f1p1, f1p2, f1p3, f2p0, f2p1, f2p2, f2p3 ]

Q: (B,12,H,D)

K: (B,12,H,D)

```
attention_scores = attention_scores.masked_fill(mask == 0, -1e9)
```

# Causal: mask future tokens

```
1 seq_len = 12
2 mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1)
3 mask
4
```

f0p0, f0p1, f0p2, f0p3, f1p0, f1p1, f1p2, f1p3, f2p0, f2p1, f2p2, f2p3

```
f0p0, [[0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
f0p1,  [0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
f0p2,  [0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
f0p3,  [0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
f1p0,  [0., 0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1.],
f1p1,  [0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1., 1.],
f1p2,  [0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1.],
f1p3,  [0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
f2p0,  [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1.],
f2p1,  [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1.],
f2p2,  [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
f2p3   [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```python
def make_video_causal_mask(F, H, W, B=1, num_heads=1, device='cpu'):
    S = F * H * W  # total number of tokens
    patches_per_frame = H * W
    frame_idx = torch.arange(S, device=device) // patches_per_frame  # [S]
    allowed = torch.zeros(S, S, dtype=torch.bool)
    # for i in range(S):         # loop over queries
    #     for j in range(S):     # loop over keys
    #         if frame_idx[i] >= frame_idx[j]:
    #             allowed[i, j] = True
    allowed = frame_idx[:, None] >= frame_idx[None, :]  # shape: [S, S]
    mask = torch.full((S, S), float('1'), device=device)
    mask[allowed] = 0.0
    return mask[None, None, :, :].expand(B, num_heads, S, S)  # shape: [B, H, S, S
make_video_causal_mask(F=3,H=2,W=2)
```

```
((((0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
    [0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
    [0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
    [0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1.],
    [0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
    [0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
    [0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
    [0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
    [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]]])
```

# Training

Given the causal arch, let’s train it!

2 sequential stages:

1. Diffusion adaptation
2. Distillation

# Training
## Diffusion Adaptation

# Training

1. Diffusion adaptation (Causal ODE Pretraining)

   finetune the base model with causal attention masking (CausVid, Self-Forcing)

   1. Generate 16K ODE solution pairs sampled from the base model

```
clean_video = base_wan_model.generate(prompt, 50_steps)  # 50-step inference #
# Then run ODE solver to get intermediate noisy versions
ode_trajectory = [add_noise(clean_video, t=1000),      # Most noisy
                  add_noise(clean_video, t=750),
                  add_noise(clean_video, t=500),
                  add_noise(clean_video, t=250),
                  clean_video                          # t=0, no noise ]
```

# Training

1. Diffusion adaptation (Causal ODE Pretraining)

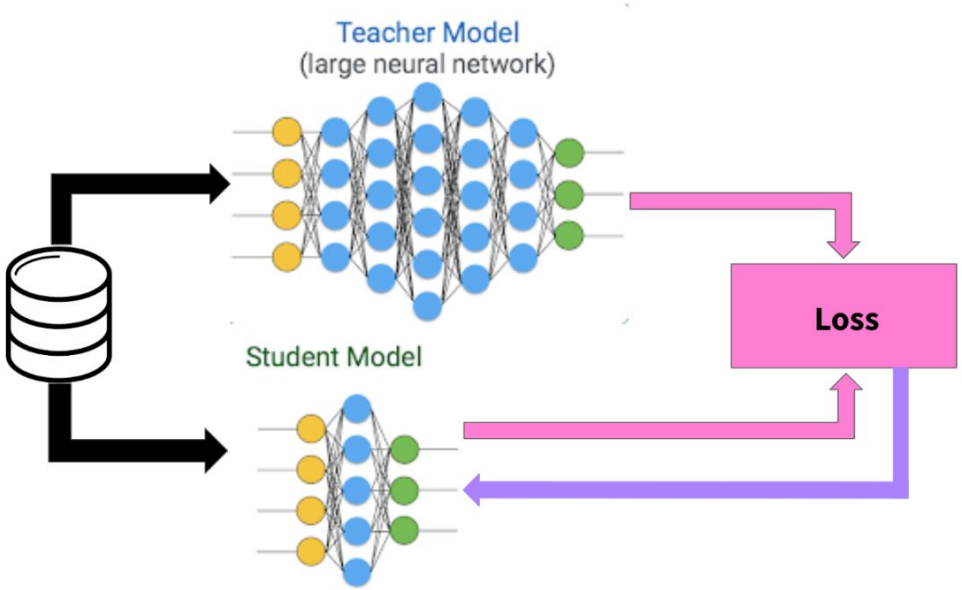
   finetune the base model with causal attention masking (CausVid, Self-Forcing)

   1. Generate 16K ODE solution pairs sampled from the base model
   2. Regression using pairs

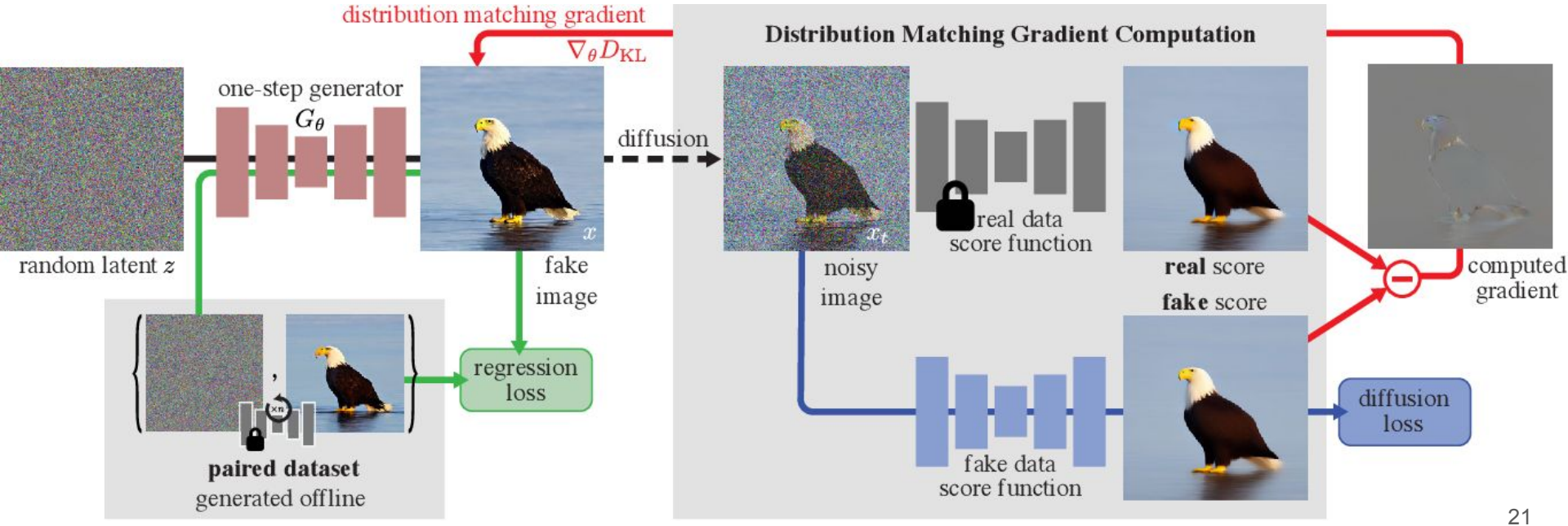
```
ode_trajectory = load_from_disk()  # [5, 21, 16, 60, 104]

# Step 1: Randomly pick ONE noisy timestep
random_timestep = random.choice([1000, 750, 500, 250])
noisy_input = ode_trajectory[index_of_random_timestep]

# Step 2: ONE forward pass through the model
# Q, K, V all from noisy_input (all 21 frames, noisy)
predicted_clean = causal_wan_student(x=noisy_input, # e.g., latent at t=500
timestep=random_timestep, # e.g., 500
text_embedding=text_emb)

# Step 3: Compare with ground truth (clean from trajectory)
ground_truth_clean = ode_trajectory[-1]  # t=0, the clean one
loss = MSE(predicted_clean, ground_truth_clean)
```

# Training
## Distillation

# Distillation



# Distribution Matching Distillation



# DMD Loss

3-component DMD loss (Both CausVid and Self-Forcing):

1. Gradient of KL (stop-grad trick)
2. Diffusion loss on generated samples (via score networks)
3. Adversarial loss from student/teacher generations (critic training)

# Standard Diffusion vs. TF. vs. DF vs. SF.

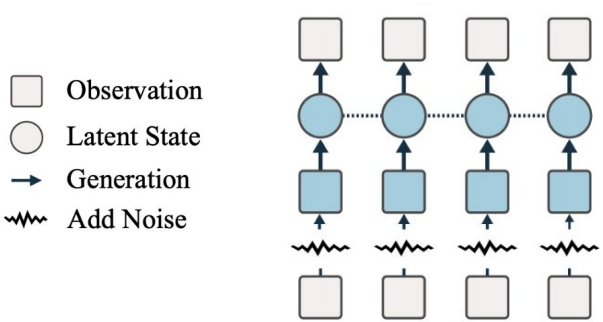
# DMD Training

Different training modes:

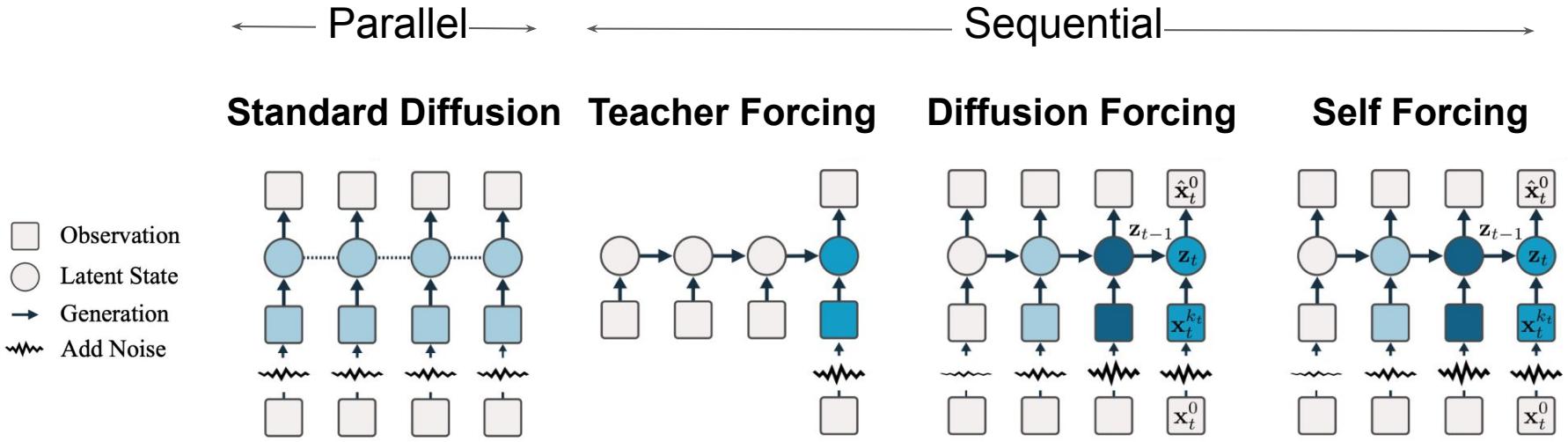
- Parallel standard diffusion
- Sequential Teacher forcing
- Sequential Diffusion forcing
- Sequential Student forcing

# Training modes

**Standard Diffusion**




| Mode | What Q come from | What K,V come from (conditioning context) |
|---|---|---|
| Standard Diffusion | Current frame/block tokens noised to *t* | all frames at the same diffusion level |




| Mode | What Q come from | What K,V come from (conditioning context) |
|---|---|---|
| Standard Diffusion | Current frame/block tokens noised to *t* | all frames at the same diffusion level |
| Teacher-Forcing | Current frame/block tokens noised to *t* | Teacher (ground-truth) context frames |
| Self-Forcing | Current frame/block tokens noised to *t* | generated previous blocks (its rollout history) |
| Diffusion-Forcing | Current frame/block tokens noised to *t* | Teacher context tokens noised to the different *t* |

# KV cache

7 blocks, each with 3 frames (21 latent, 4x → 80 frames)

**Without KV Cache**

- At block 0: compute Q,K,VQ,K,V for all frames in block 0.
- At block 1: recompute Q,K,VQ,K,V for **all previous blocks (0 + 1)**.
- At block 2: recompute for **blocks 0 + 1 + 2**, and so on.
  → **Quadratic growth** in compute with sequence length.

**With KV Cache**

- At block 0: compute Q,K,VQ,K,V; **store K,V** for this block in cache.
- At block 1:
  - Compute **only Q** and **new K,V** for current block.
  - Attend over **cached K,V** from previous blocks + current K,V.
  - Append current block's K,V to cache.

# Sparse Attention

# Faster Video Diffusion with Trainable Sparse Attention

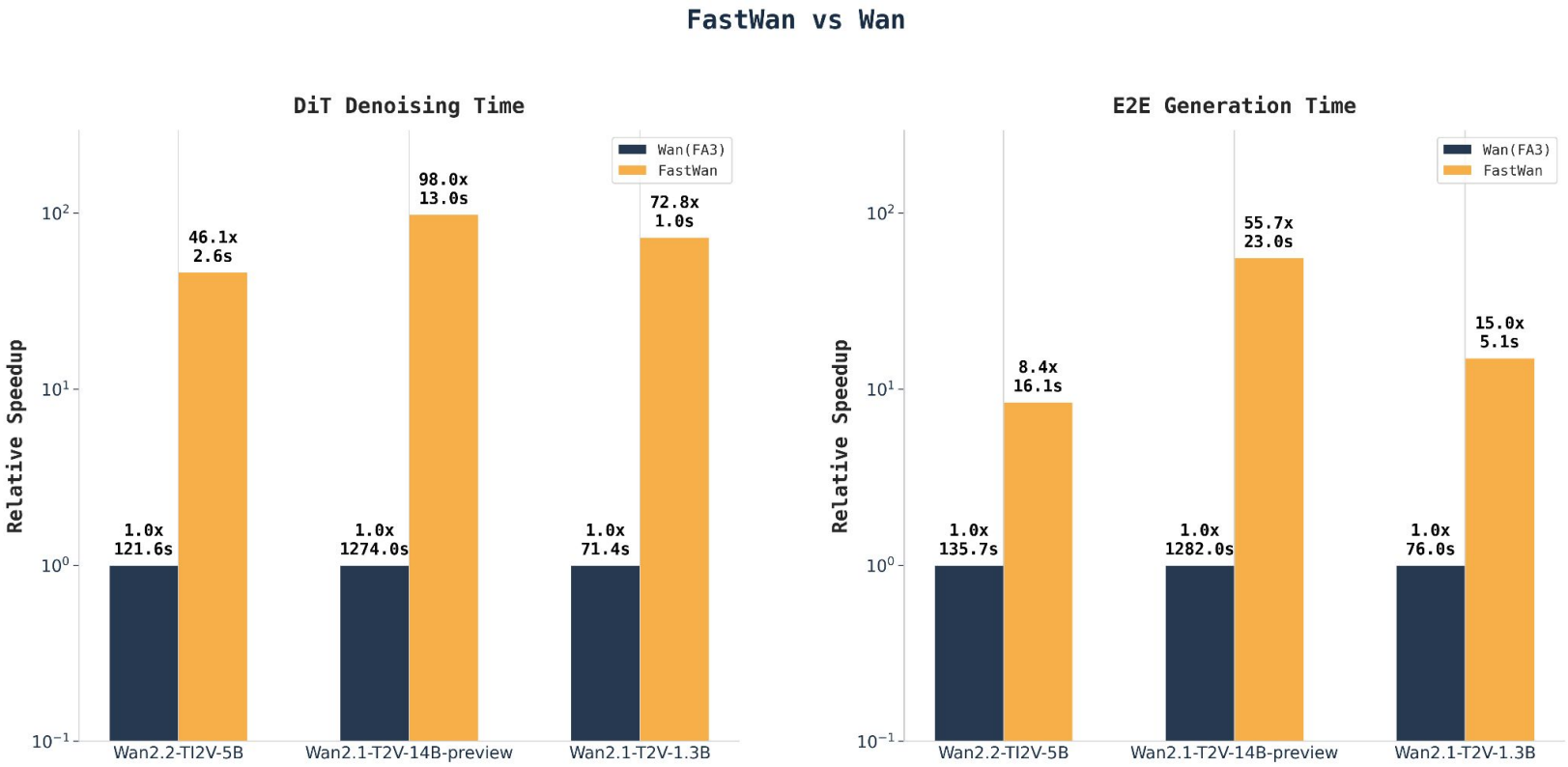
aka FastWan

# VSA

- Two main bottlenecks of video DiTs:
  - Denoising steps: 50 → 3 steps
  - Quadratic cost of attention:
    - 5sec video involves ~100K tokens at 720p
    - attention operations can eat up more than **85%** of total inference time.

      → sparse attention

# How Fast is FastWan?



# Existing Sparse Attentions fail under distillation

- Prior sparse attention (e.g., STA, SVG) prunes attention only in late denoising steps.
- Distillation (50 → 1–4 steps) removes “late stages” — redundancy vanishes.
- Existing sparse patterns fail under sub-10 step setups (confirmed in experiments).
- Sparse attention: up to **3x speedup**; distillation: **20x gains**.

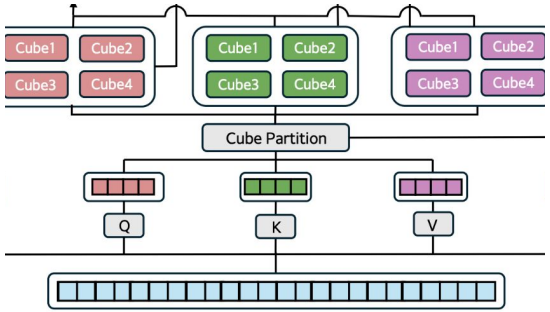
# Why VSA is distillation compatible?

- VSA dynamically identify important tokens in the sequence → replaces FlashAttention during training to learn data-dependent sparse patterns.
- **Distillation-compatible** – adapts sparse patterns as the student model learns fewer-step denoising.

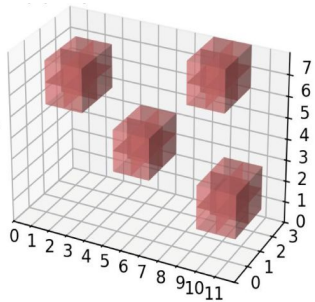
# VSA

**1) Tile tokens** – reshape Q/K/V (and gates) into spatio-temporal tiles to expose block structure.
**2) Build block reps** – pool within each tile to get coarse Qc/Kc/Vc (one vector per block).

```
// --- 1) Tile ---
q, k, v = tile(q), tile(k), tile(v)  // (B, H, token, D)

// --- 2) Mean-pool inside each block ---
q_c = q.view(B, H, token // block, block, D).mean(dim=3)  // (B, H, token//block, D)
k_c = k.view(B, H, token // block, block, D).mean(dim=3)  // (B, H, token//block, D)
v_c = v.view(B, H, token // block, block, D).mean(dim=3)  // (B, H, token//block, D)
```



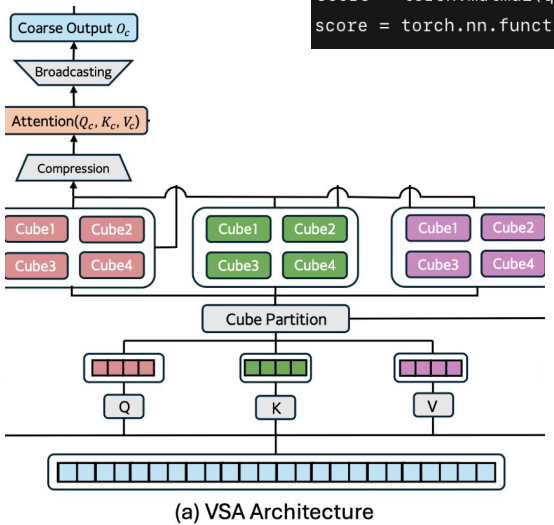
(a) VSA Architecture



(c) Cube Partition

# VSA

**1) Tile tokens** – reshape Q/K/V (and gates) into spatio-temporal tiles to expose block structure.
**2) Build block reps** – pool within each tile to get coarse Qc/Kc/Vc (one vector per block).
**3) Coarse block attention** – run cheap block-to-block attention over Qc×Kc; softmax.

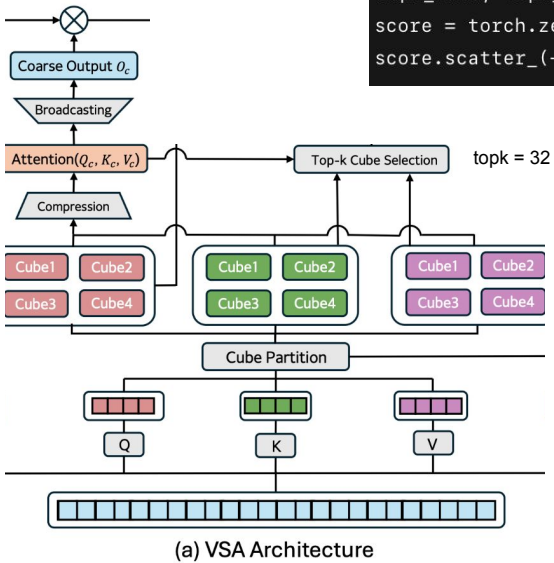


(a) VSA Architecture

```
score = torch.matmul(q_c, k_c.transpose(-2, -1)) / (D ** 0.5)  # (B, H, Nb, Nb)
score = torch.nn.functional.softmax(score, dim=-1)             # softmax over key blocks
```

# VSA

**1) Tile tokens** – reshape Q/K/V (and gates) into spatio-temporal tiles to expose block structure.
**2) Build block reps** – pool within each tile to get coarse Qc/Kc/Vc (one vector per block).
**3) Coarse block attention** – run cheap block-to-block attention over Qc×Kc; softmax.
**4) Top-k routing** – keep only the top-k key blocks per query block; zero the rest.



(a) VSA Architecture

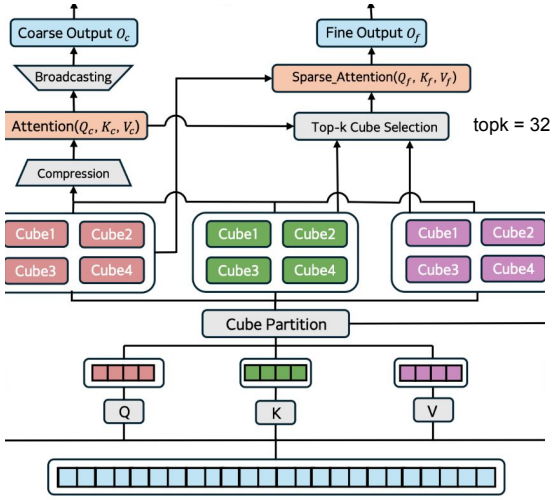
```
topk_vals, topk_idx = score.topk(topk, dim=-1)          # (B, H, Nb, topk)
score = torch.zeros_like(score)                         # (B, H, Nb, Nb)
score.scatter_(-1, topk_idx, topk_vals)                  # keep only top-k block scores
```

e.g., a patch of sky doesn't need details from distant foreground tiles
**Effect:**

- Without top-k → each block attends to **all Nb blocks** → fine stage is still $O(L^2)$.
- With top-k → each block only attends to **k blocks** ($k \ll Nb$) → fine stage cost becomes $O(L \cdot k \cdot block)$

# VSA

1) **Tile tokens** – reshape Q/K/V (and gates) into spatio-temporal tiles to expose block structure.
2) **Build block reps** – pool within each tile to get coarse Qc/Kc/Vc (one vector per block).
3) **Coarse block attention** – run cheap block-to-block attention over Qc×Kc; softmax.
4) **Top-k routing** – keep only the top-k key blocks per query block; zero the rest.
5) **Fine token attention (sparse)** – compute QK and apply the mask so attention runs only on routed pairs; softmax, then ×V.

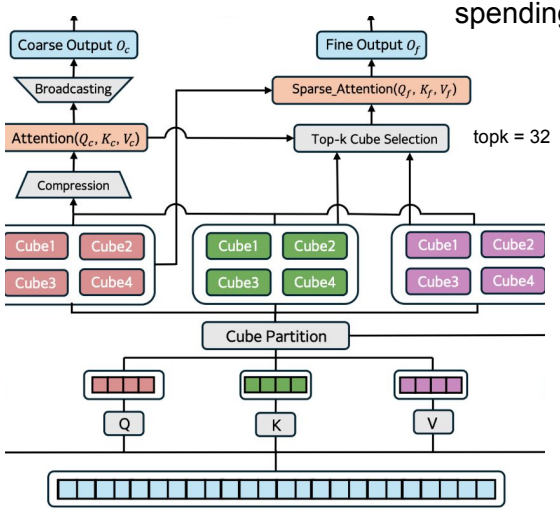


(a) VSA Architecture

# VSA

Coarse:
Give a *global but low-res* view to every block, and identify the **top-k most relevant blocks** for each query block.

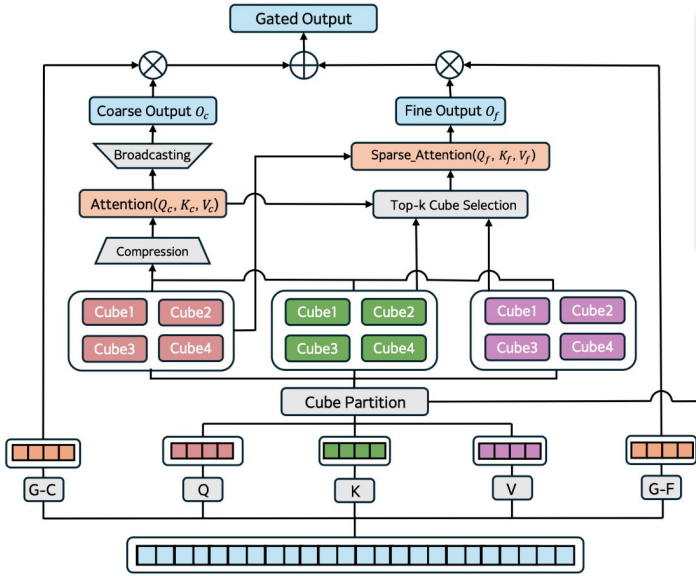
Fine:
Give a *high-res, precise* view, but **only** where coarse attention says it's worth spending compute.




(a) VSA Architecture

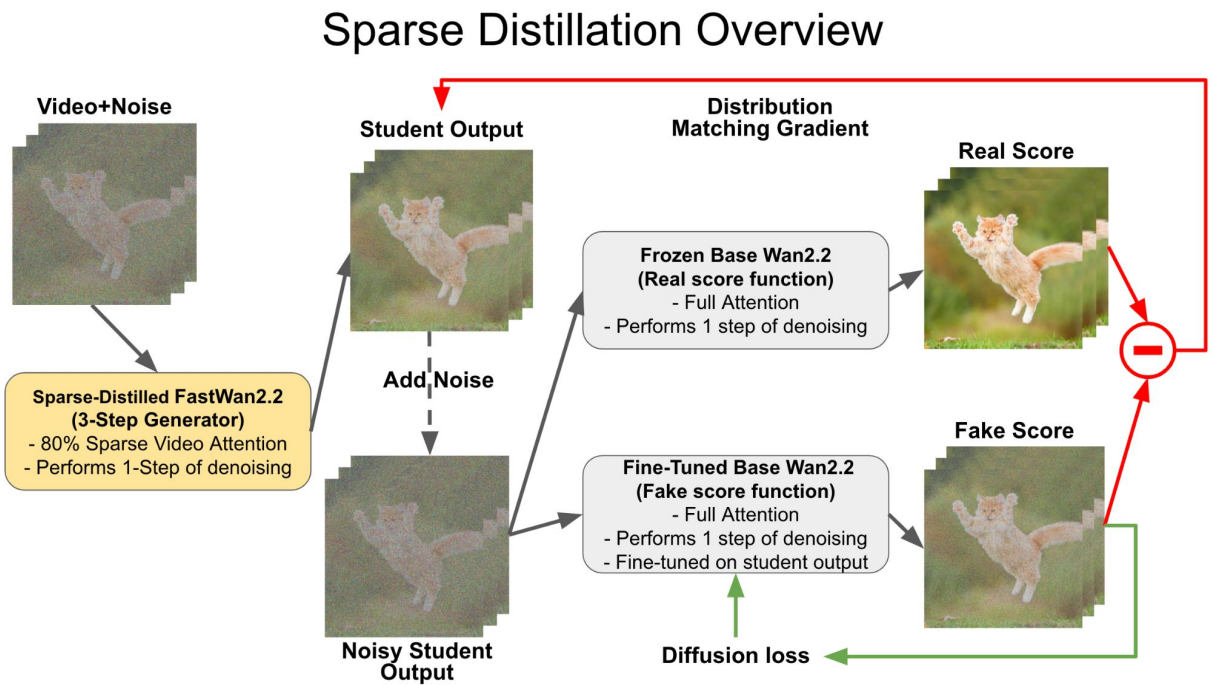
# VSA

**1) Tile tokens** – reshape Q/K/V (and gates) into spatio-temporal tiles to expose block structure.
**2) Build block reps** – pool within each tile to get coarse Qc/Kc/Vc (one vector per block).
**3) Coarse block attention** – run cheap block-to-block attention over Qc×Kc; softmax.
**4) Top-k routing** – keep only the top-k key blocks per query block; zero the rest.
**5) Fine token attention (sparse)** – compute QK and apply the mask so attention runs only on routed pairs; softmax, then ×V.
**6) Gated fusion** – blend coarse and fine outputs with learned gates (per head/token).




(a) VSA Architecture

# Distillation



# Thank you

rahim.entezari@gmail.com