GPU-vRAM Usage Estimation for Diffusion Models

Objective

Derive an analytical equation to estimate peak vRAM usage during inference for the `stable-diffusion-v1-5/stable-diffusion-v1-5` for arbitrary input image sizes.

Background

vRAM consumption during diffusion model inference differs significantly from model size on disk. Peak memory depends on:

- Model weights (fixed)
- Intermediate activations (vary with image dimensions and prompt length)
- Framework overhead (CUDA kernels, workspace buffers)
- Attention mechanism memory scaling ($O(N^2)$) with sequence length)

Where:

- `H`, `W` = input image height and width
- `prompt_length` = tokenized prompt length
- Identify any additional factors affecting vRAM

Requirements

- Analyze the architecture: Understand UNet, VAE, CLIP text encoder, and how tensors flow through the pipeline
- Account for precision: Assume `FP16` (2 bytes/parameter)
- Model fully on GPU: Ignore pipeline.enable_model_cpu_offload() in your equation
- Peak, not average: Find the stage with maximum memory allocation
- Document assumptions: Clearly state what you include/exclude (e.g., gradient storage, optimizer states)

Deliverables

- Equation with explanation of each term
- Derivation notes showing how you arrived at each component
- Validation (optional but encouraged): Compare equation predictions against actual nvidia-smi measurements using the provided test code

```
!pip install torch torchvision diffusers['torch'] transformers accelerate hf_xet
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.12/dist-packages (2.8.0+cu126)
Requirement already satisfied: torchvision in /usr/local/lib/python3.12/dist-packages (0.23.0+cu126)
Requirement already satisfied: transformers in /usr/local/lib/python3.12/dist-packages (4.57.1)
Requirement already satisfied: accelerate in /usr/local/lib/python3.12/dist-packages (1.11.0)
Requirement already satisfied: hf_xet in /usr/local/lib/python3.12/dist-packages (1.2.0)
Requirement already satisfied: diffusers[torch] in /usr/local/lib/python3.12/dist-packages (0.35.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-packages (from torch) (3.20.0)
Requirement already satisfied: typing-extensions>=4.10.0 in /usr/local/lib/python3.12/dist-packages (from torch) (4.15.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-packages (from torch) (75.2.0)
Requirement already satisfied: sympy>=1.13.3 in /usr/local/lib/python3.12/dist-packages (from torch) (1.13.3)
Requirement already satisfied: networkx in /usr/local/lib/python3.12/dist-packages (from torch) (3.5)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages (from torch) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.12/dist-packages (from torch) (2025.3.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.6.77 in /usr/local/lib/python3.12/dist-packages (from torch) (12.
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.6.77 in /usr/local/lib/python3.12/dist-packages (from torch) (1
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.6.80 in /usr/local/lib/python3.12/dist-packages (from torch) (12.
Requirement already satisfied: nvidia-cudnn-cu12==9.10.2.21 in /usr/local/lib/python3.12/dist-packages (from torch) (9.10.2
Requirement already satisfied: nvidia-cublas-cu12==12.6.4.1 in /usr/local/lib/python3.12/dist-packages (from torch) (12.6.4
Requirement already satisfied: nvidia-cufft-cu12==11.3.0.4 in /usr/local/lib/python3.12/dist-packages (from torch) (11.3.0.
Requirement already satisfied: nvidia-curand-cu12==10.3.7.77 in /usr/local/lib/python3.12/dist-packages (from torch) (10.3.
Requirement already satisfied: nvidia-cusolver-cu12==11.7.1.2 in /usr/local/lib/python3.12/dist-packages (from torch) (11.7
Requirement already satisfied: nvidia-cusparse-cu12==12.5.4.2 in /usr/local/lib/python3.12/dist-packages (from torch) (12.5
Requirement already satisfied: nvidia-cusparselt-cu12==0.7.1 in /usr/local/lib/python3.12/dist-packages (from torch) (0.7.1
Requirement already satisfied: nvidia-nccl-cu12==2.27.3 in /usr/local/lib/python3.12/dist-packages (from torch) (2.27.3)
Requirement already satisfied: nvidia-nvtx-cu12==12.6.77 in /usr/local/lib/python3.12/dist-packages (from torch) (12.6.77)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.6.85 in /usr/local/lib/python3.12/dist-packages (from torch) (12.6
Requirement already satisfied: nvidia-cufile-cu12==1.11.1.6 in /usr/local/lib/python3.12/dist-packages (from torch) (1.11.1
Requirement already satisfied: triton==3.4.0 in /usr/local/lib/python3.12/dist-packages (from torch) (3.4.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from torchvision) (2.0.2)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.12/dist-packages (from torchvision) (11.3.0)
```

```
Requirement already satisfied: importlib_metadata in /usr/local/lib/python3.12/dist-packages (from diffusers[torch]) (8.7.0
Requirement already satisfied: huggingface-hub>=0.34.0 in /usr/local/lib/python3.12/dist-packages (from diffusers[torch]) (
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.12/dist-packages (from diffusers[torch]) (2024.1
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from diffusers[torch]) (2.32.4)
Requirement already satisfied: safetensors>=0.3.1 in /usr/local/lib/python3.12/dist-packages (from diffusers[torch]) (0.6.2
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from transformers) (25.0)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.12/dist-packages (from transformers) (6.0.3)
Requirement already satisfied: tokenizers<=0.23.0,>=0.22.0 in /usr/local/lib/python3.12/dist-packages (from transformers) (
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.12/dist-packages (from transformers) (4.67.1)
Requirement already satisfied: psutil in /usr/local/lib/python3.12/dist-packages (from accelerate) (5.9.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from sympy>=1.13.3->torch) (1
Requirement already satisfied: zipp>=3.20 in /usr/local/lib/python3.12/dist-packages (from importlib_metadata->diffusers[to
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.12/dist-packages (from jinja2->torch) (3.0.3)
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests->diffuser
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests->diffusers[torch]) (3
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests->diffusers[torc
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests->diffusers[torc
```

Additional steps, I did:

1. Created a virtual envirnoment (python -m venv env)
2. Activated the env (env\Scripts\activate)
3. Changed the jupyter kernel to the env
4. Run the pip installations (above cell) in terminal
5. Loaded it google colab, used t4 gpu

Step 1 to 2 was for local setup. Since, the device wasn't able to run the full code, switched to colab to see the results.

```
import torch
from diffusers import AutoPipelineForImage2Image
from diffusers.utils import make_image_grid, load_image
import subprocess
```

```
pipeline = AutoPipelineForImage2Image.from_pretrained(
    "stable-diffusion-v1-5/stable-diffusion-v1-5", torch_dtype=torch.float16, variant="fp16", use_safetensors=True
)
pipeline = pipeline.to("cuda" if torch.cuda.is_available() else "cpu")
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), se
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
model_index.json: 100%                                                541/541 [00:00<00:00, 55.9kB/s]
Fetching 15 files: 100%                                               15/15 [00:16<00:00,  1.03s/it]
merges.txt:       525k/? [00:00<00:00, 2.82MB/s]
config.json:      4.72k/? [00:00<00:00, 66.3kB/s]
config.json: 100%                                                   617/617 [00:00<00:00, 8.22kB/s]
preprocessor_config.json: 100%                                        342/342 [00:00<00:00, 5.79kB/s]
special_tokens_map.json: 100%                                         472/472 [00:00<00:00, 44.9kB/s]
scheduler_config.json: 100%                                           308/308 [00:00<00:00, 27.5kB/s]
text_encoder/model.fp16.safetensors: 100%                                   246M/246M [00:04<00:00, 35.3MB/s]
safety_checker/model.fp16.safetensors: 100%                                 608M/608M [00:13<00:00, 43.7MB/s]
tokenizer_config.json: 100%                                           806/806 [00:00<00:00, 13.6kB/s]
config.json: 100%                                                   743/743 [00:00<00:00, 17.8kB/s]
vocab.json:       1.06M/? [00:00<00:00, 15.5MB/s]
config.json: 100%                                                   547/547 [00:00<00:00, 30.2kB/s]
unet/diffusion_pytorch_model.fp16.safete(…): 100%                           1.72G/1.72G [00:15<00:00, 141MB/s]
vae/diffusion_pytorch_model.fp16.safeten(…): 100%                           167M/167M [00:04<00:00, 35.2MB/s]
Loading pipeline components...: 100%                                  7/7 [00:00<00:00,  9.78it/s]
`torch_dtype` is deprecated! Use `dtype` instead!
```

```
# Uncomment this if you have limited GPU vRAM (although, this assignment can be done without any GPU use!)
# pipeline.enable_model_cpu_offload()
```

```
# remove following line if xFormers is not installed or you have PyTorch 2.0 or higher installed
# pipeline.enable_xformers_memory_efficient_attention()
```

```
def get_gpu_memory_via_nvidia():
    result = subprocess.run(
        ['nvidia-smi', '--query-gpu=memory.used', '--format=csv,noheader,nounits'],
        capture_output=True, text=True
    )

    return int(result.stdout.strip()) / 1024
```

```
# prepare image
img_src = [{
    "url": "./data/balloon--low-res.jpeg",
    "prompt": "aerial view, colorful hot air balloon, lush green forest canopy, springtime, warm climate, vibrant foliage,
}, {
    'url': "./data/bench--high-res.jpg",
    'prompt': "photorealistic, high resolution, realistic lighting, natural shadows, detailed textures, lush green grass,
}, {
    'url': "./data/groceries--low-res.jpg",
    'prompt': "cartoon style, bold outlines, simplified shapes, vibrant colors, playful atmosphere, exaggerated proportion
}, {
    'url': "./data/truck--high-res.jpg",
    'prompt': "Michelangelo style, Renaissance painting, classical composition, rich earthy tones, detailed brushwork, div
}]

results = list()
```

```
# Nvidia smi before execution
before_execution = get_gpu_memory_via_nvidia()
print("GPU memory before execution: ", before_execution)
```

```
GPU memory before execution:  2.7265625
```

```
# Getting the all the necessary details for the vram calculator pipeline
f_parameters = []
for i, test_src in enumerate(img_src):
  init_image = load_image(test_src.get('url'))

  # Getting dimensions
  h, w = init_image.size[1], init_image.size[0]

  # Getting prompt length or text tokens
  prompt = test_src.get('prompt')
  tokens = pipeline.tokenizer(prompt, return_tensors="pt")
  prompt_length = tokens['input_ids'].shape[1]

  f_parameters.append({
      'heights': h,
      'widths': w,
      'prompt_lengths': prompt_length
  })
```

```
# This for loop is meant to demonstrate that the models' vRAM usage depends
# on Image-size and prompt length (among other factors). You may observe the
# vRAM usage while the model is running by executing the following command
# in a separate terminal and monitoring the changes in vRAM usage:
#    ```shell
#    watch -n 1.0 nvidia-smi
#    ```
#
# You may modify this for loop according to your needs.
for _src in img_src:
    init_image = load_image(_src.get('url'))
    prompt = _src.get('prompt')

    # pass prompt and image to pipeline
    image = pipeline(prompt, image=init_image, guidance_scale=5.0).images[0]
    results.append(make_image_grid([init_image, image], rows=1, cols=2))
```

```
results[0].show()
```

```
100%                                                40/40 [00:04<00:00, 10.93it/s]
100%                                                40/40 [09:55<00:00, 14.73s/it]
100%                                                40/40 [00:11<00:00,  3.61it/s]
100%                                                40/40 [02:52<00:00,  4.18s/it]
```

```
# Nvidia smi after execution
after_execution = get_gpu_memory_via_nvidia()
print("GPU memory after execution: ", after_execution)
```

```
GPU memory after execution:  9.76171875
```

Your Task

Derive a formula:

My understanding of Stable diffusion architecture:

- Input = {Prompt, Image}
- Image -> VAE encoder -> latent image
- Latent image + noise -> noisy latent
- Noisy latent -> UNET encoders -> downsampling + extract useful imformation/features
- Prompt -> Text encoder (word2vec) -> Embeddings
- Output of UNET encoder + Embeddings -> UNET decoder -> Predicted noise (Upsampling) // Magic happens here
- Noisy latent - predicted noise = new latent image (by denoising)
- New latent image -> VAE decoder -> Completely new image

Phase 1: Base VRAM -> How much VRAM is needed to just load the model?

- All the parameters of the model gets loaded into VRAM.
- Therefore, calculating the total parameters and weights is required.
- Base VRAM = combining total parameters with the weights.
- Since, it's fp16 (float 16), each number is going to take 2 bytes. Thus, multiply with 2 to get the bytes used.

```
fp16_bytes = 2
```

```
# calculating base vram
def get_base_vram():
    unet_params = sum(component.numel() for component in pipeline.unet.parameters())
    vae_params = sum(component.numel() for component in pipeline.vae.parameters())
    text_encoder_params = sum(component.numel() for component in pipeline.text_encoder.parameters())

    total_params = unet_params + vae_params + text_encoder_params

    weights = total_params * fp16_bytes
    return weights
```

Phase 2: Activations (variable memory) -> How much VRAM will be required based on the input?

1. VAE encoder -> depends on image (height and width)
2. UNET -> depends on image and prompt (height, width, prompt_length)
3. VAE decoder -> depends on image (height and width)

Phase 2.1. VAE Encoder

1. Amount of memory needed to store in the GPU (VRAM).

- During the processing, the VAE stores up some temporary tensors.

- Temporary tensors does take up memory which is estimated to be 3 to 4 times the input memory.

2. Memory needed for compressed image or latent image. Eg:- (512 x 512 x 3) image -> (64 x 64 x 4).

- Compression = 8x (512 / 64 = 8), total 64x
- And channel is changed from 3 to 4 for storing for essential information.

```
def latent_values(height, width):
    latent_height = height // 8
    latent_width = width // 8

    latent_memory = latent_height * latent_width * 4 * fp16_bytes
    return latent_height, latent_width, latent_memory
```

```
def vae_encoder_memory(height, width):
    input_memory = height * width * 3 * fp16_bytes
    temp_memory = input_memory * 3
    _, _, latent_memory = latent_values(height, width)

    total_memory = input_memory + temp_memory + latent_memory
    return total_memory
```

Phase 2.2. UNET

For the image:

- Input will be latent values from vae encoder.
- Initial layers take up lot of memory.
- As we go down memory reduces and channel goes up. Reason being information gets compressed.
- So, I decided to take average channels of the layers (Calculating each one is too complex.)

A typical layer structure (got from hugging face github) { 320, 640, 1280 }

Average = (320 + 640 + 1280) / 3 = 746.667 => 745

- Similar to VAE, unet also uses temporary tensor memory, assuming it to be 3x.

For the prompt:

- prompt length -> text tokens
- attention matrix is calculated.
- Attention matrix - mapping words to the image regions. (Letting the LLM know what part of the image to be influenced by which word).

```
def unet_memory (height, width, prompt_length):
    latent_h, latent_w, _ = latent_values(height, width)
    channel_activations = latent_h * latent_w * 745 * fp16_bytes
    channel_memory = channel_activations * 3

    attention_matrix = latent_h * latent_w * prompt_length * fp16_bytes

    total_memory = channel_memory + attention_matrix
    return total_memory
```

Phase 2.3 VAE Decoder

- Input is the denoised image from unet, which is basically the latent image.
- However, only the image will be different while dimensions will be same as the latent image which was fed to UNET.
- Then, it produces a new image which will have same dimenisions as the original image.
- And during the creation of new image, it generates temporary tensors assuming it to be 3x memory.

```
def vae_decoder_memory(height, width):
    _, _, latent_memory = latent_values(height, width) # input
    output_memory = height * width * 3 * fp16_bytes # output image
    temp_memory = output_memory * 3

    total_memory = latent_memory + output_memory + temp_memory
    return total_memory
```

```
# VRAM calculation pipeline
```

```
# VRAM calculation pipeline
def f(h: int, w: int, prompt_length: int, **kwargs):
    """
    :param h: height of input image in pixels
    :param w: width of input image in pixels
    :param prompt_length: length of input prompt in number tokens generated after tokenizing the input-prompt.
    :param kwargs: any additional factors needed for this computation (this is for your use)
    """
    # Write your code here!

    base_mem = get_base_vram()
    encod_mem = vae_encoder_memory (h, w)
    unet_mem = unet_memory(h, w, prompt_length)
    decod_mem = vae_decoder_memory (h, w)

    peak_mem = max(encod_mem, unet_mem, decod_mem)

    """
    Assuming a 20% extra memory.
    So, total_memory_percentage = 100 + 20 = 120%
    overhead_memory_value = 120 / 100 = 1.2
    """
    overhead_fact = 1.2

    total_mem = (base_mem + peak_mem) * overhead_fact
    total_mem_gb = total_mem / (1024**3)

    return total_mem, total_mem_gb
```

Validation

```
total_mem_req = 0

for params in f_parameters:
    _, each_mem_req = f(params['heights'], params['widths'], params['prompt_lengths'])
    total_mem_req = total_mem_req + each_mem_req
```

```
print("My function prediction: ", total_mem_req)
print("Nvidia SMI calculations: ", after_execution)

print("Difference in error: ", total_mem_req - after_execution)
```

```
My function prediction:  10.087232579290866
Nvidia SMI calculations:  9.76171875
Difference in error:  0.32551382929086614
```

Tips

- Although no GPU is needed to accomplish this task (analyze code/architecture)
- Use PyTorch documentation and model architecture inspection

Evaluation Criteria

- Correctness: Formula accounts for major memory consumers
- Completeness: All image-dependent and prompt-dependent factors identified
- Rigor: Derivation shows understanding of PyTorch memory model and diffusion architecture
- Clarity: Equation is readable and well-documented