DSCI 421, Final Project

Implementing a Generative AI Model on GPU

Eddie McGowan

Department of Electrical and Computer Engineering at Lehigh University

wem220@lehigh.edu

Date: May 10th, 2024

Abstract

This project had two parts: the first was to learn about diffusion models by completing the NVIDIA "Generative AI for Diffusion Models" course; and the second was to deploy a diffusion model from this course on a high-performance computer (HPC). Diffusion models take text as the input and build a corresponding image from pure noise. It works by encoding the text and image descriptions and then training a model to recognize the similarities in the encodings. This model uses a U-Net architecture, which slowly transforms an image into noise and then rebuilds the image. This process teaches the model how to generate a new image from noise. Diffusion models have many optimization strategies which include normalization, GELU activation function, einops pooling, sinusoidal position embedding, and using deeper networks. I completed the NVIDIA course and received the certification, which I attached along with the assessment code. The code I deployed on the HPC was from module 5 “CLIP”, which allows for the generation of flower images. The code is primarily PyTorch-based and uses a pre-trained model to associate the text and image encoding [1]. Most of the model is built into the code. I faced challenges running PyTorch due to the HPC configuration. Thanks to the help of Prof. Zheng, Ruoyu, and David, I was able to get these issues resolved, and I shared my learnings with the class. Once my code ran, I compared the results on the HPC against the Nvidia AWS course. The NVIDIA course produced similar results in a faster time. In addition, I compared my diffusion model against the learning text-to-image model DALL-E 3.0 by Open AI. DALL-E produced higher-quality results than my diffusion model. However, I that neither model generates flower images that could be confused with a natural photograph. I enjoyed this project and hope to use diffusion models in the future. Thank you, Professor Zheng, and Ruoyu for all your help this semester! I enjoyed this class and learned a lot.

Introduction

Overview: The goal of this project was to learn about Generative AI text-to-image diffusion models and see how these models can be deployed. This project consisted of two parts. The first part of this project was to complete the NVIDIA "Generative AI for Diffusion Models” course, which taught about diffusion

models and provided coding labs to execute these models. I completed this course and assessment. I attached my assessment code and certification to my submission. I have summarized the theory behind this course later in this Introduction Section. The second part of this project was to deploy a text-to-image Generative AI model on Lehigh's Electrical and Computer Engineering Department's high-performance computer (HPC). Afterward, I compared the performance of the Generative AI model on the NVIDIA AWS course against the HPC in both CPU and GPU. Also, I detailed how I deployed the code on the HPC and the challenges I faced. Before this project, I had no experience with AI algorithms or packages. I enjoyed the freedom of this project as I was able to apply topics from this class (including parallel computing and convolutional neural networks) to my area of interest. This project gave me a greater appreciation of larger Generative AI models such as Dall-E and stable diffusion. The project took around forty hours to complete. This course contained dozens of equations and multiple derivations. I will be focusing on the key equations and theory. The contents of this paper are entirely created by me, and references are used where applicable. Chat GPT was used to refine my grammar [1].

Generative AI Overview (GANs): Generative AI diffusion models are often built using general adversarial networks (GANs), which have two components, namely, a discriminator and a generator. A GAN architecture can be seen below in Figure 1. A discriminator uses a convolutional neural network (CNN), similar to the CNN created in Project 1, to classify images as real or fake. A fake image represents a picture created by the generator. A generator is an architecture like a CNN but in reverse. This network is "adversarial" as these two components compete. The loss function of the discriminator is based on its ability to correctly identify real and fake images. The loss function of the generator is based on its ability to trick the discriminator into believing a fake image is real. To implement a GAN, a U-net architecture is often used (as seen below in Figure 2). In the U-net architecture, the discriminator is called an encoder. the generator is called a decoder, and the output of the encoder is linked to the input of the decoder. The benefit of a U-net is training images are broken down using a CNN and then accurately reconstructed at a

high detail. In the context of this project, U-nets are used so the model can "understand" the "real" input images during model training and then generate appropriate "fake" images when prompted by the user [1].

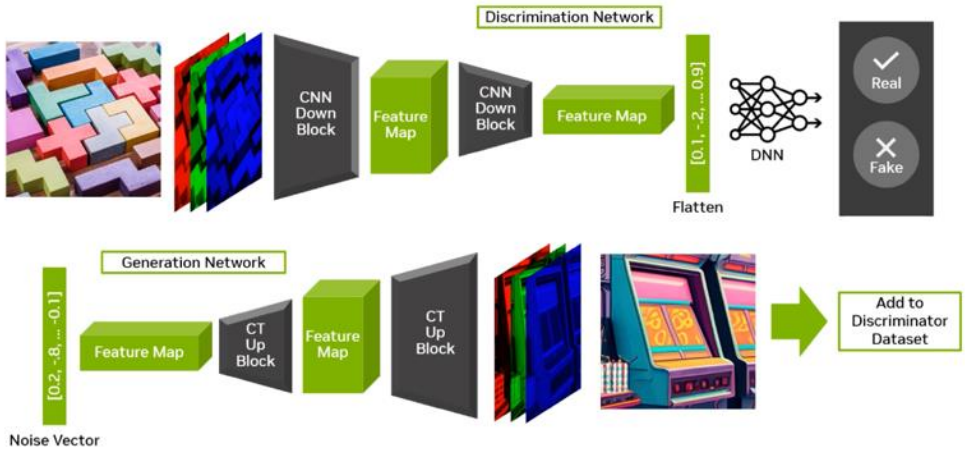


Figure 1 – Generative Adversarial Network architecture. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

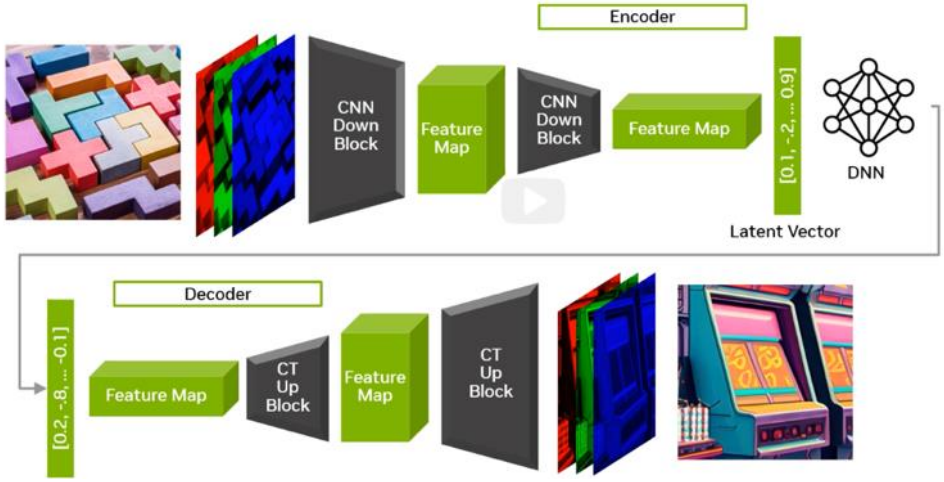


Figure 2 – U-Net autoencoder architecture. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

U-Nets: In Figure 3, an alternative view of the U-net architecture can be seen. During each CNN "block" of the encoder, the image size decreases and the number of channels increases in the feature map. This is called the down block and helps extract features from the training images. Each channel can represent different textures, colors, or edges. This is accomplished using the same kernel multiplication as Project 1. In contrast, during each CNN "block" of the decoder, the image size increases and the number of channels decreases in the feature map This is called the up-block and is used to generate a single appropriate detailed image. The decoder uses transposed convolution. In transposed convolution, the image's pixel values are multiplied against a kernel. However, the input image is expanded before the convolution begins, allowing for the output image to be larger than the input. For example, in Figure 4 below, the original image was a 3x3 grid of 1's and 0's in the shape of an X. The stride of 2 expands the image to a 7x7 by adding 0s between each row and column in the image. Using standard convolution, the image size reduces to 6x6,

which is still larger than the initial 3x3 starting image. Mean squared error (MSE) is commonly used for the loss function of the U-Net, which measures the difference in pixel values between the decoder's generated image and the original image. In transposed convolution, there are differences in how terms are applied. In traditional convolution, stride refers to the step size across pixels before multiplying the kernel against a set of pixel values. In transposed convolution, stride represents expanding the image by adding rows and columns of 0's. In traditional convolution, padding adds rows and columns around the image border. In transposed convolution, padding removes rows and columns around the image. Transposed convolution has an additional feature called output padding, which adds back the rows removed by padding. This feature is useful to set output image size [1].

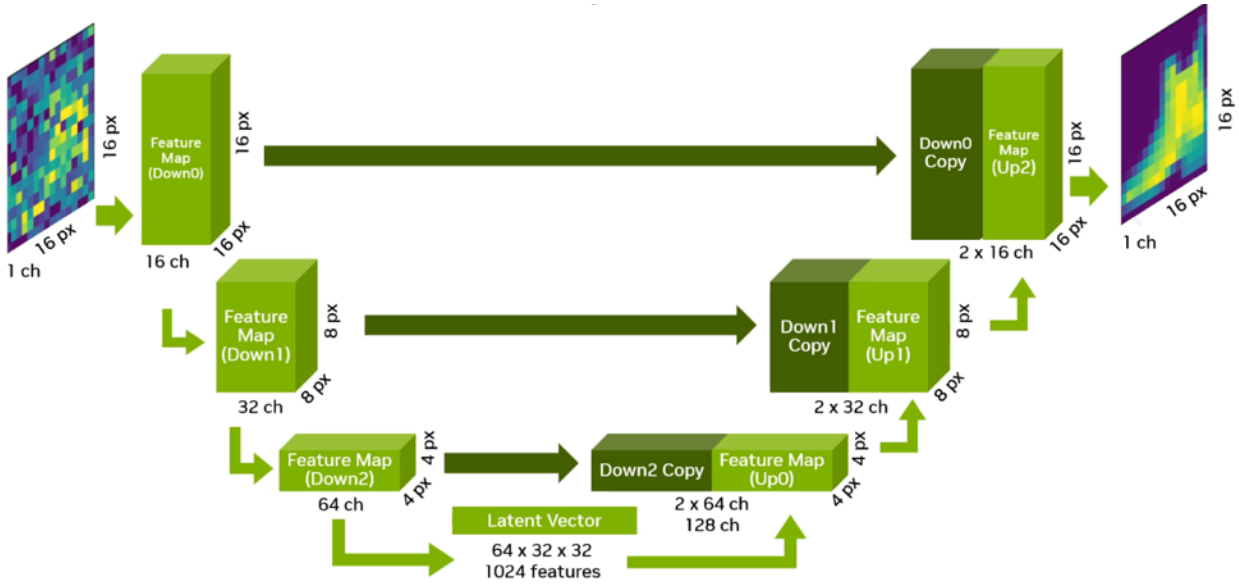


Figure 3 – U-Net convolution feature map. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

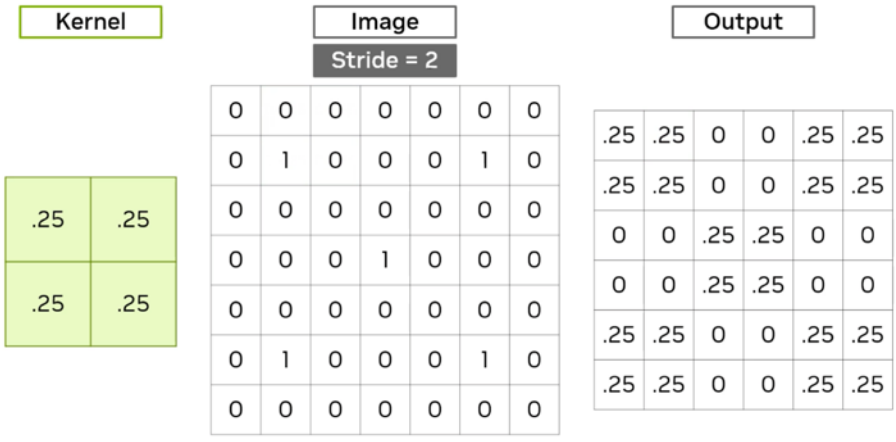


Figure 4 – Transposed convolution example. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

Diffusion Models: GANs are an excellent model for creating new images from training images. However, GANs face challenges for text-to-image models as they struggle to generate a new image from pure noise with no direct reference to the original image. On the other hand, this is an area of strength for diffusion models, which also use a U-Net style architecture and have two components: forward diffusion and reverse diffusion. In forward diffusion, noise is added iteratively. In reverse diffusion, this noise is iteratively removed. By starting with a real image and then blurring it with noise, the model is better able to recreate the image using pure noise as a starting point. The loss function for diffusion models is also commonly MSE. In forward diffusion, the noise added follows a blurring schedule. To initially preserve the features of the image, less noise is added in the earlier blurring iterations, then noise is increased during later iterations. The noise added commonly follows the equation seen below in Equation 1. $q(X_t, X_{t-1})$ represents the difference in pixel value between iterations. β_T represents the noise variance schedule. X_t represents the current pixel value, and X_{t-1} represents the previous pixel value. "I" represents a randomly chosen value in the normal distribution. The different time iterations are represented using matrix time embedding. For example, in time step 5, a value of 5 is added to all values in the matrix. This can be seen below in Figure 6 [1].

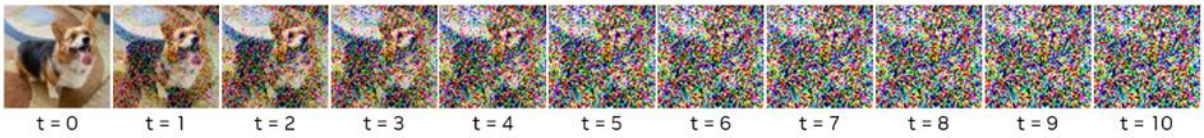


Figure 5 –Example of Diffusion Model. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = N(\mathbf{x}_t; \sqrt{1-\beta_t} \cdot \mathbf{x}_{t-1}, \beta_t \cdot \mathbf{I})$$

Equation 1 –Noise added in forward diffusion equation. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

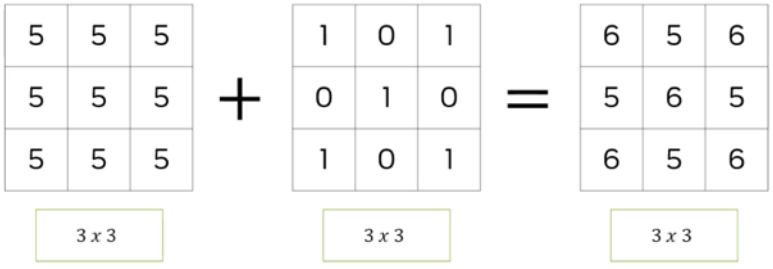


Figure 6 –Time embedding in matrix for time = 5. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

In reverse diffusion, the noise per iteration is estimated and then subsequently removed. In Equation 2, the noise removal equation can be seen. This equation on the left uses a similar format and variables as equation

1. The equation on the right is based on Bayes Theorem, where α is equal to β minus 1. Bayes theorem tells the probability of an event occurring. ε_t represents the amount of noise added. μ represents the mean distribution of the data at the previous timestamp. Using a neural network, which is trained on the forward diffusion dataset, the noise per iteration (ε_t) can be properly estimated. This allows noise to be removed after each iteration. However, the noise removal is limited by the training dataset. For instance, the model cannot remove noise to reveal a flower if the model was only trained on dog images [1].

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = N(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \cdot \mathbf{I}) \quad \tilde{\mu}_t = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x_t} - \frac{1-\alpha_t}{\sqrt{1-\overline{\alpha_t}}}\epsilon_t\right)$$

Equation 2 –Noise removed in reverse diffusion equation (left) and Bayers theorem on the right. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

Optimizations: To optimize diffusion models there are many strategies including normalization, GELU activation function, einops pooling, sinusoidal position embedding, and deeper networks. First, normalization smoothens the image output and addresses the checkerboard problem. In this problem, the generated image contains a pattern that resembles a checkerboard. This is caused when the kernel size is not a multiple of the image size, causing some pixels to be more processed than others in the model training. The two main types of normalization are batch normalization and group normalization. Batch normalization converts all the output of an image kernel to a z-score before assigning a pixel value. Group normalization converts the output across the image channels to a z-score before assigning a pixel value. Second, GELU reduces the risk of dead neurons in diffusion models. The GELU function appears similar to a RELU function, as seen below in Figure 7. However, the GELU function is curved, causing the derivative to be non-zero and keeping the neuron active. This allows the neural network to capture more complex patterns. Third, einops is a library that allows more information to be preserved during pooling. The standard pooling method, max pooling, only takes the maximum value contained in a window. Using einops, values are split across channels. Max pooling is performed on each channel. By increasing channels as the feature matrix reduces in size, the a greater percentage of the initial image features can be preserved. Fourth, sinusoidal position embedding prevents the model from recognizing time steps as a continuous variable by encoding time as an angle on the unit circle. As discussed in the diffusion section, each time step has a different amount of blurring. If the model identifies time as a continuous variable, it may wrongly assume the blurring per iteration is also continuous. By associating time as an angle of the unit circle, time steps are placed in a sequence. Lastly, adding more convolution layers in each down block and up block improves the generated image. This increases the number of steps to transform from pure noise to a generated image, resulting in a more appropriate detailed final image [1].

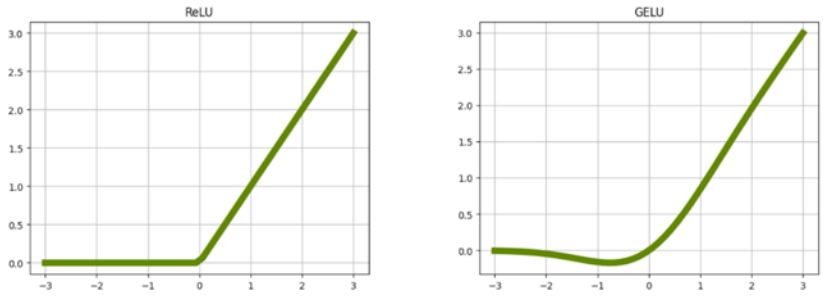


Figure 7 –RELU (left) vs GELU (right) activation function. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

Adding Context: To improve the accuracy of the generated image, context is embedded in the training images in the diffusion model. This is accomplished by one-hot encoding the image category in the feature map. This allows the model to recognize differences between categories, such as shoes and shirts. Once all context features are added, individual context features are randomly dropped during forward diffusion using a Bernoulli mask, which allows the neural network to link the image features to the context encoding. After encoding, two sets of noise are added: one with image context and one without image context. The noise with the image context exaggerates the features of that category. It is created by averaging the pixel values of images in that category. The noise without context is useful in extracting image texture and background information. The balance between the noises with and without the context is represented by the hyperparameter "W", which is set during training. This can be seen in Figure 8 with a fashion dataset. When W is largely positive, the context will get a high weight, and the training category will be generated. The generated image will closely resemble the average of the training images in that category. When W=0, the image's category has little influence on the generated image. When W is negative, the model places less emphasis on the image context, resulting in more varied results including images in different categories than the training image. For example, in Figure 8 row 1 column 2, the context is pants, but the image produces a shoe. In this text-to-image model, the ideal W is around one, which allows for slight variance in image generation while considering the image context [1].

Figure 8 –Impact of image context blurring in Image Generation. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

CLIP: To create a text-to-image model, a text context embedding needs to be linked to an image context embedding. For the text context embedding, a pre-trained model called Contrastive Language-Image Pre-Training (CLIP) version ViT-B/32 is used in this project. To link the text and image embedding, cosine similarity is used. As seen below in Figure 9, cosine similarity maps the vectors associated with the text and image embedding to the unit circle. This is achieved by dividing each component by the vector's magnitude. The components on the unit circle are multiplied against each other by taking the dot product. The result is the cosine between the two vectors. A cosine of 1 represents the context of the image and text vectors being aligned. A cosine of 0 represents the context of the image and text vectors having no relation. A cosine of -1 represents the context of the image and text vectors having a contrasting relationship. In the project, CLIP trained the model to associate the training image embedding with the text embedding. As seen below in Figure 10, CLIP calculates the cosine similarity between all text inputs and the training images. CLIP uses cross-entropy to maximize where the cosine similarity is the highest for the image. In the model, the cosine similarity is calculated between the training images and CLIPs pre-trained 512 features [1].

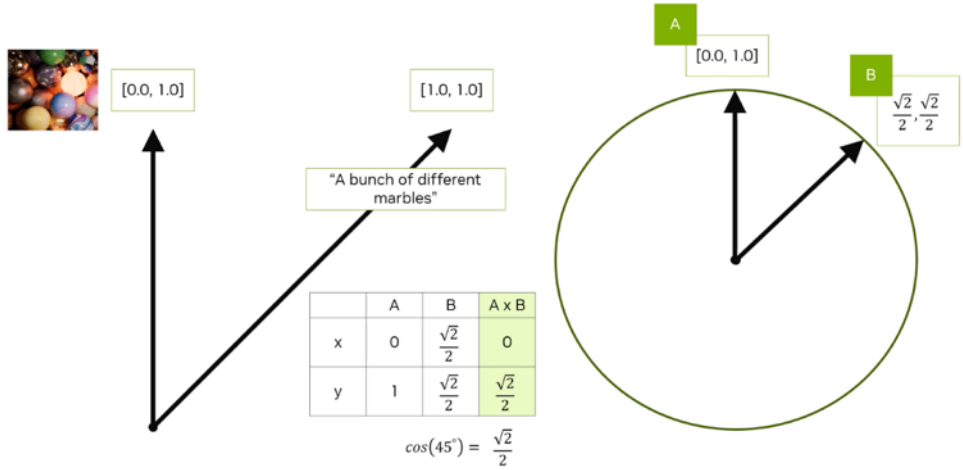


Figure 9 –Cosine similarity alignment between a picture of marbles and a text description. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

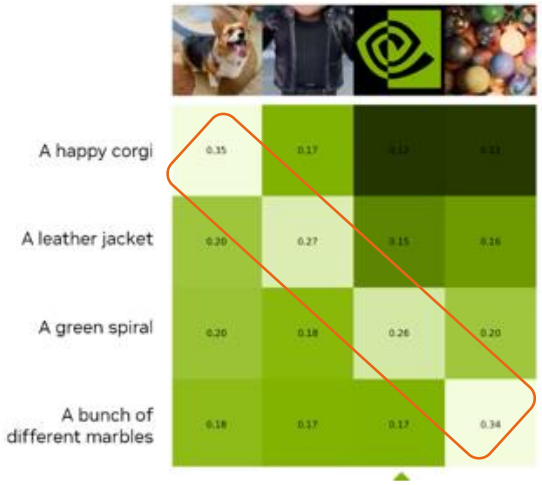


Figure 10 –Cosine similarity alignment between a picture of marbles and a text description. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

Procedure

For this report, I completed the following tasks to deploy a text-to-image model on the HPC.

1. Completed the lessons, examples, and certification assessment for the NVIDIA course "Generative AI with Diffusion Models" [1].
2. Read through the reference links for this course to better understand the content.
3. Recorded the CPU and GPU used. The GPU was found using nvidia-smi, and the CPU was found using pip install py-cpuinfo, then import cpuinfo; info = cpuinfo.get_cpu_info(); for key, value in info.items(): print(f"{key}: {value}") [1,2].
4. Downloaded the code and training images from module "05_Clip." In addition, I recorded all the prerequisites to execute the code including package names/versions, and compiler version. To

determine the packages used, I read through the import statements. To determine the version of each package, "!conda list" and "!pip list" were run. To determine the compiler version, the following command below was run [3].

   a. import subprocess
      gcc_version = subprocess.getoutput('gcc --version | head -n1 | awk \'{print $NF}\')
      print("GCC Version:", gcc_version)

5. All the code and images were placed in the same local folder and transferred to the HPC using the Linux command prompt scp -r /path/user###@ece-hpc##.cc.lehigh.edu:/path.
6. In the HPC, upgraded the compiler using the prompt "scl enable devtoolset-11 – bash". This command needed to be approved by David before it could be used on the HPC.
7. Started a virtual Python environment (created in project 1) using the prompt conda active python_environment [4].
8. Installed the necessary packages numpy, torch, matplotlib, ipython, torchvision, einops, and clip. Torch and torch vision GPU were installed using the prompt "conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia". The remaining packages were installed using the command "conda install package_name". For packages where conda install was not available, "pip install package_name" was used instead.
   a. Torch and torchvision CPU were installed using "conda install pytorch torchvision torchaudio".
   b. The Nvidia course used an up-to-date version of these packages [1].
   c. The Python packages csv, math, and glob were also used in the deployed Generative AI model. However, these are pre-installed in Python, so they did not need to be installed.
9. Started a Jupyter local server using the prompt "Jupyter notebook --no-browser --port=XXXX." Copied the provided server URL. In a new tab, mapped the Jupyter port instance to the local machine using the command "ssh -L ####: localhost:#### user@ece-hpc##.cc.lehigh.edu".
10. Entered the URL in the browser to navigate to Jupyter. Navigated to the file and run all the cells on CPU and GPU [4].
11. Compared the runtime between the NVIDIA course and the HPC CPU and GPU.
12. Compared the model performance against OpenAI's DALL-E 3.0
13. Reported my experience with the course and what I learned about diffusion models.
14. Detailed my experience moving deploying the code on the HPC and the challenges I faced.

Generative AI with Diffusion Models: This course contained six modules, six coding labs, eight videos, and a coding assessment. Module 1 discussed the U-net architecture and how they are used in Generative AI diffusion models. The coding lab for this module had two components. The first was to add and remove

noise from the Fashion MNIST dataset using the U-Net architecture. The second was to start with random noise to generate one of the items from the fashion MNIST dataset. This dataset contains 28 pixel by 28 pixel images of different fashion items, including t-shirts/tops, trousers, pullovers, dresses, coats, sandals, shirts, sneakers, bags, and ankle boots [6]. Noise, which follows the normal distribution, is added to the training images and the model removes the noise. Afterward, the model attempts to create the MNIST Fashion images from random noise. Due to the simplicity of the model, the resulting images are unclear. In module two the same noise removal is re-run using a diffusion model. Due to additional iterations of adding and removing noise, the resulting images were somewhat clearer. In module three, the fashion MNIST diffusion model is optimized using normalization, GELU, einops, sinusoidal position embedding, and increased model depth. Adding these optimizations resulted in clearer generated images. In module four, categorical embedding, Bernoulli masks, and context/no-context diffusion were added to the fashion MNIST model. This resulted in the best results. After this example, these strategies were applied to colored images of daisies, sunflowers, and roses. These images were cropped on the colored portion of the flower. The color caused the model to take longer to train. In the fifth module, a text-to-image algorithm was created on the flower dataset using the CLIP model ViT-B/32 [1].

Results and Analysis

Q1.1. What did you learn from the reference links?

The Nvidia course referenced many formulas and functions used in the image generation models. However, the course does not cover the derivations of these formulas, or how these functions can be used. The reference links explain this context. As there were dozens of reference links, I included the most useful links below.

GitHub repository for fashion MNIST dataset: Link
Documentation of torchvision (part of pytorch): Link
Transposed convolution in PyTorch: Link
Mathematics of diffusion models: Link
Implementing diffusion models in GANs: Link
Bayes Theorem: Link
Batch and group normalization: Link
Einops documentation: Link
Sinusoidal position encoding: Link
Context embedding for diffusion guidance: Link

CLIP GitHub repository: Link

Q1.2. What are the applications of text-to-image diffusion models?

Text-to-image diffusion models have applications in product development, marketing, and graphic design. For product development, these models can convert product descriptions into prototype images, thereby speeding up the prototyping process. For marketing, these models can create advertisements based on a brand strategy and a target demographic, which accelerates content creation while ensuring relevance. For graphic designers, these models can quickly render 3D environments, which allows the designers to take more creative risks while also improving development time.

Q1.3. What interesting knowledge did you learn in this project?

I was surprised by the computational intensity of text-to-image models. I have used DALL-E and stable diffusion, which takes up to a minute to generate complex images. I assumed by running a simpler model on AWS and HPC, I would have a similar runtime. However, I found that even simple models can take 30 minutes to 15 hours to run. This realization provided me with a greater appreciation for the computation resources used by companies such as OpenAI. In addition, I enjoyed learning about the optimization techniques used by these models, including normalization, GELU activation function, einops pooling, sinusoidal position embedding, and using deeper networks, and how these small optimizations nevertheless create large improvements in the resulting images.

Q1.4. What CPU and GPU were used in this project?

To find the CPU, the following commands were run: pip install py-cpuinfo, then import cpuinfo; info = cpuinfo.get_cpu_info(); for key, value in info.items(): print(f"{key}: {value}"). To find the GPU, !nvidia-smi was used.

In the course, “NVIDIA Generative AI with Diffusion Models” X86_64 was used for the CPU, which had 4 cores and 2.8 GHz frequency. An NVIDIA A10G was used for the GPU, which has a power capacity of 300W and a memory usage of 23028MiB (see Figure 11). Two NVIDIA GeForce RTX 2080 Ti were used, each GPU has a power capacity of 250W and a memory usage of 11264MiB (see Figure 12). On the HPC, X86_64 was used for the CPU, which had 8 cores and a 3.7 GHz frequency. The full CPU information can be seen in Appendix Figures 1 and 2.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A10G         On   | 00000000:00:1E.0 Off |                    0 |
|  0%   27C    P0    59W / 300W |    314MiB / 23028MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Figure 11 –GPU used in the NVIDIA course. Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:05:00.0 Off |                  N/A |
| 31%   48C    P0    78W / 250W |      0MiB / 11264MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce ...  Off  | 00000000:09:00.0 Off |                  N/A |
| 36%   47C    P0    52W / 250W |      0MiB / 11264MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure 12 –GPUs used in the HPC

Q1.5 Explain the model

As a preface, the model is long and utilizes numerous package functions. For simplicity and clarity, I am going to explain the code at a high level. In the appendix, I have included a flow chart of the model. The majority of the code was taken from the NVIDIA "Generative AI with Diffusion Models" course. I made three changes. First, thanks to the help of Ruoyu, I added save checkpoints in my model training which allowed my code to run without any issues or interruptions. However, if my code was interrupted, I would have been able to load the current iteration of my model training to avoid any lost work. These checkpoints are a built-in feature of pyspark with the torch.save() and torch.load functions [5]. Second, IPython was used to show images before they were processed by the model. IPython was causing difficulties on the HPC. I swapped IPython with matplotlib to show the images, which did not affect the model's performance. Lastly, I did not load the NVIDIA header image into the HPC as this image is not needed for the model. I am thankful for NVIDIA providing this course, and their work has been cited appropriately throughout this paper.

In the encoding section of the model, four sets of packages and files were imported. The first set were packages that were necessary for model development: csv, glob, numpy, torch, ipython, torchvision, einops, math, and clip. The second set were packages needed to visualize the results: matplotlib, PIL, torchvision, and text wrap. The third set were user-defined libraries: other_utils, ddpm_utils, and UNet_utils. Other_utils was used to show images in a grid and save images in a gif, and ddpm_utils was used for forward and reverse diffusion. ddpm stands for "denoising diffusion probabilistic models." UNet_utils was used to generate the U-net. The fourth set was the CLIP model, which was loaded and set as the model type "ViT-B/32" and the number of features was set as 512. Once the packages were loaded, the model was set as CPU or GPU. Both CPU and GPU were used in separate iterations.

With the packages imported, the flower image dataset was then loaded. This dataset contained daises, sunflowers, and roses. Before running the full model, the model first runs an example using one daisy, rose, and sunflower image, and a corresponding text description of each image. To link the text explanation to the image context, the image and text encoding were created. To create the image encoding, the image was converted to a stack and then converted to a tensor using clip_preprocess(). These tensors were converted to an encoding using clip_model.encode_image(). Creating the text encoding was similar to creating the image encoding. The text describing each image was converted into a numeric tensor using clip.tokenize(). These tensors were converted to an encoding using clip_model.encode_text(). Both vector encodings were normalized to have a one-unit magnitude. For confirmation, this approach worked and every combination of image and text encoding was compared. To achieve this comparison, the repeat() and repeat_interleave commands were used to duplicate the text and image encoding to ensure each combination was performed. To measure the similarity between the text and image encoding, cosine similarity was used. The dot product between these two vectors was calculated, which creates a cosine similarity score by vector. The sum of the cosine similarity scores was calculated to get the total similarity between the text and image. The higher the similarity, the closer the relation between the image and text. The cosine similarity scores were shown in a grid to confirm and visualize the text-to-image alignment, see Figure 13

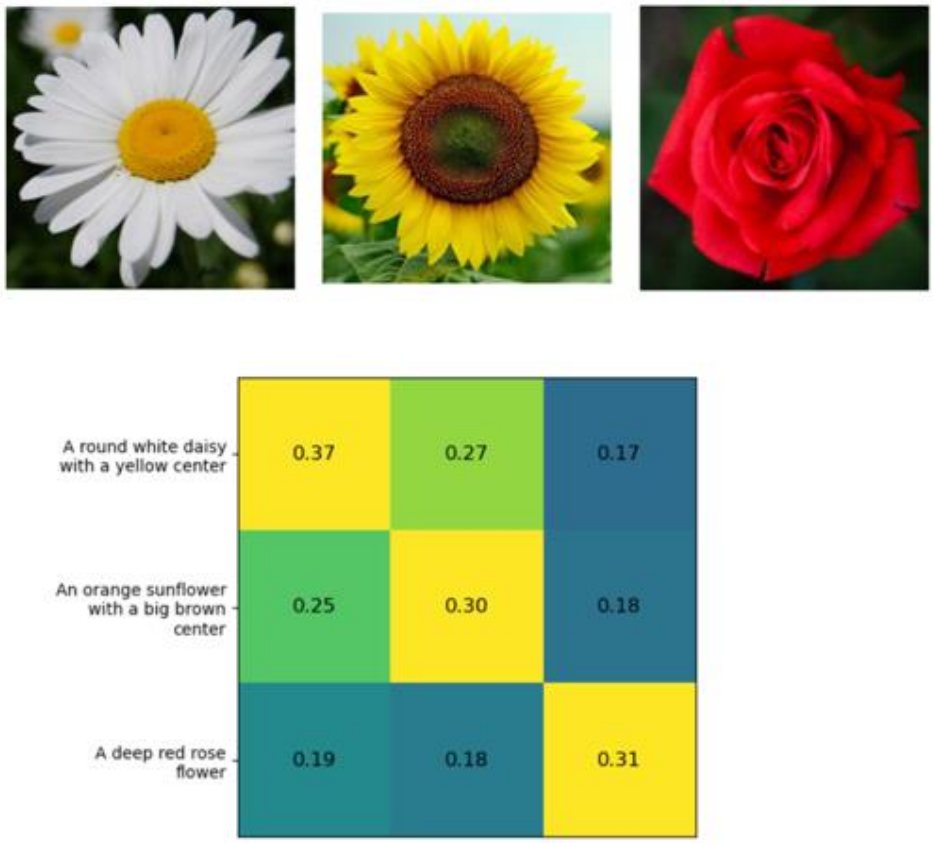


Figure 13 – Similarity matrix for the images and text used in the model. Utilized code from "NVIDIA Generative AI with Diffusion Models"[1]

After confirming the accuracy of the image-to-text associations, these steps were repeated for the entire image training dataset, which included 450 images of daisies, 382 images of roses, and 332 images of sunflowers. Instead of creating individual tensors, a list was created that contains all the tensors for each of the training images. First, all of the image file paths were saved in a list using the glob package. Second, a csv was created (called clip.csv) which would hold the resulting tensors. One at a time, each image was opened, converted to a, and then added as a new list element/row in the clip.csv file as an embedding.

Once the embeddings were set, the images were preprocessed in a custom class called "My_Dataset" which contains an initialization function, a get item function, and a length function. This class created two lists; one for the image (img) and one for the embedding (labels). For each image in the csv dataset, a set of tensor transformations was performed in "pre_transforms". The transformation steps included resizing the image to 32 pixels by 32 pixels, normalizing the vector embedding to have a one-unit magnitude, and scaling each embedding element to be between -1 and 1. These changes ensure that the vectors have a constant magnitude and range. Once normalized, the get item function calls a transformation called "random_transforms". This transformation randomly crops an image then rescales the image to 32 pixels

by 32 pixels, and randomly decides to horizontally flip the image, which helps improve training image variance. Further, the get item function checks if the image has a CLIP encoding. If the encoding is present, then the image and the corresponding encoding are loaded into "img" and "label" respectively. If the encoding is not present, then the image encoding is created and then loaded into "img" and "label". The output is the preprocessed images and the labels. Also, a function called dataloader is used to define how images are iterated during training. It states that 128 images will randomly be chosen for each training batch.

A U-Net model was created called "model_flowers". First, a blurring schedule was set to increase between 0.0001 and 0.02 over 400-time steps. Second, a U-Net was created with 400-time iterations, three image channels, eight vector dimension time embedding, and 512 clip features. This u-net has three downsampling iterations which increase the channels from 3 to 256 (first downsampling), to 256 (second downsampling), and to 512 (third downsampling). The total number of parameters was printed, which is 44,900,355.

The model was trained on 100 epochs, with each epoch processing batches of 128 images. The neural network learning rate was set as 1e-4. Training used a nested for loop which iterated over each epoch and image. To increase image variability, the blurring rate corresponding to random time iteration is applied to the image. The neural network loss was calculated along with the corresponding gradient. The model's neural network parameters were adjusted accordingly. After every epoch, the model loss was printed. After every 5 epochs, an image from the model training was saved. This section took the longest time, so I added pytorch checkpoints to this section and a time function to see how long the model took to train.

Lastly, using the trained model, text descriptions were provided, and the model encoded the text and d the corresponding images in a 3 by 7 grid. Each column represented a different prompt. The images in the columns were ordered from least amount to most training of training (see Figure 14). A GIF of the generated images being created from noise was saved and attached to my submission. The text descriptions provided for Figure 14 were: "A round white flower," "A round orange sunflower with a big brown center", and "A red rose bud".

Figure 14 – Output from text-to-image model. A round white flower (left), A round orange sunflower with a big brown center (center), and "A red rose bud (right). Increase in training going from top to bottom. Utilized code from "NVIDIA Generative AI with Diffusion Models"[1]

Q1.6. Compare the runtime to for example utilizing the AWS and the HPC.

On AWS, the training code had a CPU time of 9 minutes 56 seconds and a Wall time of 9 minutes 55 seconds (see Figure 15). The AWS, testing code had a training CPU time of 15.8 seconds and a Wall time of 15.7 seconds (see Figure 15). On the HPC GPU, the training code had a CPU time of 12 minutes 55 seconds and a Wall time of 2 hours 10 minutes 3 seconds (see Figure 15). The HPC GPU testing code had a CPU time of 16.3 seconds and a Wall time of 15.7 seconds (see Figure 15). The '%%time' function was used to get these times. Based on my discussion with Ruoyu, CPU time is the average time the code takes to execute on the CPU cores. Wall time is the total time the code took to execute, which includes waiting time. The Wall time was likely longer on the HPC than AWS, because code needed to wait for resources to become available on the HPC. With either fewer users or more cores on the HPC, the HPC and AWS Wall times would be closer. In addition, the NVIDIA course had a more powerful CPU. In addition, I ran the code on the CPU, and it had a time of 15 hours. I was not aware of the '%%time' function at that time, so I used a clock. The code started at noon and finished at 3 am).

```
CPU times: user 7min 57s, sys: 1min 58s, total: 9min 56s
Wall time: 9min 55s
```

```
CPU times: user 6min 51s, sys: 6min 3s, total: 12min 55s
Wall time: 2h 10min 3s
```

Figure 15 – AWS GPU training time (left). HPC GPU time (right). Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

```
CPU times: user 15.5 s, sys: 237 ms, total: 15.8 s CPU times: user 9.87 s, sys: 6.45 s, total: 16.3 s
Wall time: 15.4 s                                  Wall time: 15.7 s
```

Figure 16 – AWS GPU training time (left). HPC GPU time (right). Reprinted from "NVIDIA Generative AI with Diffusion Models"[1]

Q1.7. Compare the model results against DALL-E 3.0?

The leading text-to-image model is DALL-E 3.0, created by OpenAI. This model currently requires a premium subscription but is available for free on minitoolai.com. I provided the same prompts used in the diffusion model (see Figure 14 for model images) to DALL-E 3.0. The images generated by DALL-E can be seen in Figure 17. These images took around 15 seconds to generate [7]. I found the images created by the diffusion model to be inadequate. For example, the petals on the rose and sunflower in Figure 14 do not look natural. I found the images created by DALL-E 3.0 to be of high quality, but they appear too perfect. The daisy and sunflower have near-perfect symmetry, and none of the images contain imperfections. Overall, I find that neither model generates flower images that could be confused with a natural photograph.

Figure 14 – Output from DALL-E 3.0. A round white flower (left), A round orange sunflower with a big brown center (center), and "A red rose bud (right) [7].

Q1.8. What problems did you encounter in these courses and how did you solve them?

I faced challenges running PyTorch on the HPC. To solve this challenge, I worked with Professor Zheng, Ruoyu, David, and Lehigh Technology Services. Thanks to this help, I got the PyTorch to run. I created and attached a PowerPoint explaining how to install PyTorch on the HPC, which I presented in class. Fixing this set of changes took over 10 hours. First, when I ran PyTorch, my code would get an error "-std=c++17" needs to be enabled. We found that the HPC compiler needed to be upgraded from gcc 4.8.5 to gcc 11. Once activated, I was able to run PyTorch on the CPU. Second, while PyTorch ran on the CPU, it was slow and my laptop would log out of the HPC before the code was completed. To solve this issue, the CPU code was run on one of the desktops in Packard Room 332, and checkpoints were implemented in the model training. The desktop was able to complete the CPU code without logging out of the HPC, so the checkpoints were not needed. Third, the GPU code would not recognize the GPUs on the HPC. The

CPU and GPU versions of PyTorch cannot be installed in the same environment. To install the GPU version of PyTorch, the CPU version of PyTorch needed to be removed along with its supporting packages torchvision and torchaudio. In addition, the HPC uses CUDA 12.0 and the current version of PyTorch uses CUDA 12.1, which prevented the latest version of PyTorch from running on the HPC. PyTorch CUDA 11.8 was installed on the HPC to solve this issue, which recognizes the HPC GPUs [8]. By upgrading the compiler, downloading an older version of PyTorch, and running the code on a Lehigh desktop, the GPU code was able to run without issues.

Another challenge was that the Nvidia course provided general details on the theory of diffusion models but did not provide much guidance on how the code worked. To write this paper and run the code on the HPC, I needed to read the documentation on the key functions and diffusion modeling.

Conclusion

This project had two parts: the first was to learn about diffusion models by completing the NVIDIA "Generative AI for Diffusion Models" course; and the second was to deploy a diffusion model from this course on a high-performance computer (HPC). Diffusion models take text as the input and build a corresponding image from pure noise. It works by encoding the text and image descriptions and then training a model to recognize the similarities in the encodings. This model uses a U-Net architecture, which slowly transforms an image into noise and then rebuilds the image. This process teaches the model how to generate a new image from noise. Diffusion models have many optimization strategies which include normalization, GELU activation function, einops pooling, sinusoidal position embedding, and using deeper networks. I completed the NVIDIA course and received the certification, which I attached along with the assessment code. The code I deployed on the HPC was from module 5 “CLIP”, which allows for the generation of flower images. The code is primarily PyTorch-based and uses a pre-trained model to associate the text and image encoding [1]. Most of the model is built into the code. I faced challenges running PyTorch due to the HPC configuration. Thanks to the help of Prof. Zheng, Ruoyu, and David, I was able to get these issues resolved, and I shared my learnings with the class. Once my code ran, I compared the results on the HPC against the Nvidia AWS course. The NVIDIA course produced similar results in a faster time. In addition, I compared my diffusion model against the learning text-to-image model DALL-E 3.0 by Open AI. DALL-E produced higher-quality results than my diffusion model. However, I that neither model generates flower images that could be confused with a natural photograph. I enjoyed this project and hope to use diffusion models in the future.

References

1. Nvidia. "Generative AI with Diffusion Models" Nvidia Deep Learning Institute, 2024, https://learn.nvidia.com/courses/course?course_id=course-v1:DLI+S-FX-14+V1&unit=block-v1:DLI+S-FX-14+V1+type@vertical+block@a53a6ca795474e579a6f1f722016b8df
2. (2024). Getting processor information in Python. Stack Overflow. https://stackoverflow.com/questions/4842448/getting-processor-information-in-python
3. sysconfig — Provide access to Python's configuration information. (2024). Python Documentation. https://docs.python.org/3/library/sysconfig.html
4. Wang, Ruoyu. "How to Access Lehigh ECE HPC workstations." docs.google.com, 2024, docs.google.com/document/d/1VcOwDb7VrSI5X2ZzLyYp1KjMpoWEd-Op-xIk4aEBs0Q/edit
5. *Saving and loading a general checkpoint in PyTorch — PyTorch Tutorials 2.3.0+cu121 documentation*. (2024). Pytorch.org. https://pytorch.org/tutorials/recipes/recipes/saving_and_loading_a_general_checkpoint.html
6. zalandoresearch. (2017). fashion-mnist/README.md at master · zalandoresearch/fashion-mnist. GitHub. https://github.com/zalandoresearch/fashion-mnist/blob/master/README.md
7. Free DALL-E 3 - Image Generator. (2021). Minitoolai.com. https://minitoolai.com/dall-e-3/
8. Start Locally. (2024). PyTorch. https://pytorch.org/get-started/locally/

Appendix

```
import cpuinfo
info = cpuinfo.get_cpu_info()
for key, value in info.items(): print(f"{key}: {value}")
```

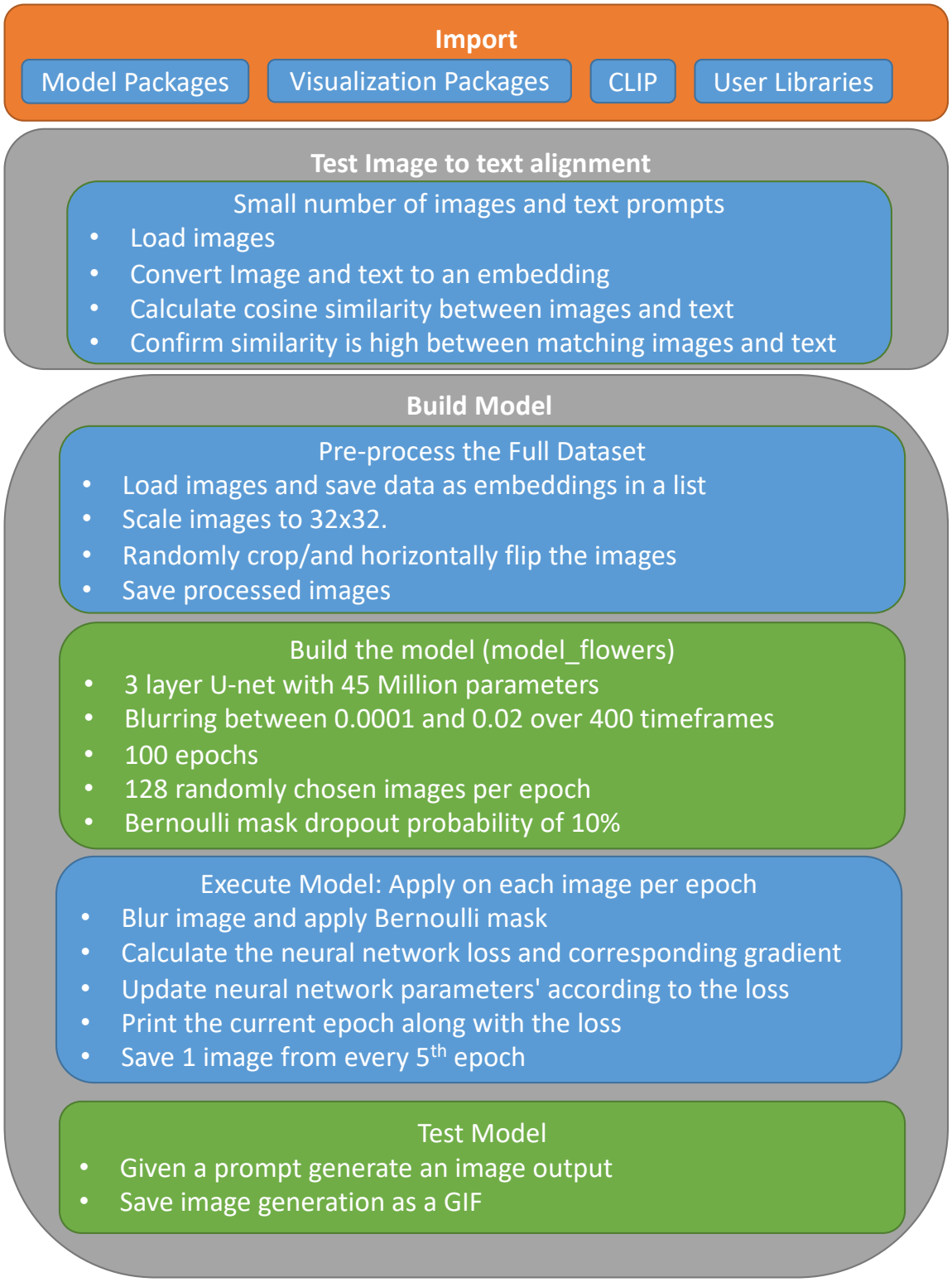
```
python_version: 3.10.6.final.0 (64 bit)
cpuinfo_version: [9, 0, 0]
cpuinfo_version_string: 9.0.0
arch: X86_64
bits: 64
count: 4
arch_string_raw: x86_64
vendor_id_raw: AuthenticAMD
brand_raw: AMD EPYC 7R32
hz_advertised_friendly: 2.8000 GHz
hz_actual_friendly: 2.8000 GHz
hz_advertised: [2799978000, 0]
hz_actual: [2799978000, 0]
model: 49
family: 23
flags: ['3dnowext', '3dnowprefetch', 'abm', 'adx', 'aes', 'aperfmperf', 'apic', 'arat', 'avx', 'avx2', 'bmi1', 'bmi2', 'clf
lush', 'clflushopt', 'clwb', 'clzero', 'cmov', 'cmp_legacy', 'constant_tsc', 'cpuid', 'cr8_legacy', 'cx16', 'cx8', 'de', 'e
xtd_apicid', 'f16c', 'fma', 'fpu', 'fsgsbase', 'fxsr', 'fxsr_opt', 'ht', 'hypervisor', 'ibpb', 'ibrs', 'lahf_lm', 'lm', 'mc
a', 'mce', 'misalignsse', 'mmx', 'mmxext', 'movbe', 'msr', 'mtrr', 'nonstop_tsc', 'nopl', 'npt', 'nrip_save', 'nx', 'osxsav
e', 'pae', 'pat', 'pclmulqdq', 'pdpe1gb', 'pge', 'pni', 'popcnt', 'pse', 'pse36', 'rdpid', 'rdpru', 'rdrand', 'rdrnd', 'rds
eed', 'rdtscp', 'rep_good', 'sep', 'sha', 'sha_ni', 'smap', 'smep', 'ssbd', 'sse', 'sse2', 'sse4_1', 'sse4_2', 'sse4a', 'ss
se3', 'stibp', 'syscall', 'topoext', 'tsc', 'tsc_known_freq', 'vme', 'vmmcall', 'wbnoinvd', 'xgetbv1', 'xsave', 'xsavec', '
xsaveerptr', 'xsaveopt']
l3_cache_size: 524288
l2_cache_size: 1048576
l1_data_cache_size: 65536
l1_instruction_cache_size: 65536
l2_cache_line_size: 512
l2_cache_associativity: 6
```

Appendix Figure 2 –CPU used in the "NVIDIA Generative AI with Diffusion Models"[1]

```
import cpuinfo
info = cpuinfo.get_cpu_info()
for key, value in info.items(): print(f"{key}: {value}")
```

```
python_version: 3.11.5.final.0 (64 bit)
cpuinfo_version: [9, 0, 0]
cpuinfo_version_string: 9.0.0
arch: X86_64
bits: 64
count: 8
arch_string_raw: x86_64
vendor_id_raw: GenuineIntel
brand_raw: Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz
hz_advertised_friendly: 3.7000 GHz
hz_actual_friendly: 1.7985 GHz
hz_advertised: [3700000000, 0]
hz_actual: [1798510000, 0]
stepping: 1
model: 79
family: 6
flags: ['3dnowprefetch', 'abm', 'acpi', 'adx', 'aes', 'aperfmperf', 'apic', 'arat', 'arch_perfmon', 'avx', 'avx2', 'bmi1', 'bmi2', 'bts', 'cat
_l3', 'cdp_l3', 'clflush', 'cmov', 'constant_tsc', 'cqm', 'cqm_llc', 'cqm_mbm_local', 'cqm_mbm_total', 'cqm_occup_llc', 'cx16', 'cx8', 'dca',
'de', 'ds_cpl', 'dtes64', 'dtherm', 'dts', 'eagerfpu', 'epb', 'ept', 'erms', 'est', 'f16c', 'flexpriority', 'flush_l1d', 'fma', 'fpu', 'fsgsba
se', 'fxsr', 'hle', 'ht', 'ibpb', 'ibrs', 'ida', 'intel_ppin', 'intel_pt', 'intel_stibp', 'invpcid', 'invpcid_single', 'lahf_lm', 'lm', 'mca',
'mce', 'mmx', 'monitor', 'movbe', 'msr', 'mtrr', 'nonstop_tsc', 'nopl', 'nx', 'pae', 'pat', 'pbe', 'pcid', 'pclmulqdq', 'pdcm', 'pdpe1gb', 'pe
bs', 'pge', 'pln', 'pni', 'popcnt', 'pse', 'pse36', 'pts', 'rdrand', 'rdseed', 'rdt_a', 'rdtscp', 'rep_good', 'rsb_ctxsw', 'rtm', 'sdbg', 'se
p', 'smap', 'smep', 'smx', 'spec_ctrl', 'ss', 'ssbd', 'sse', 'sse2', 'sse4_1', 'sse4_2', 'ssse3', 'stibp', 'syscall', 'tm', 'tm2', 'tpr_shado
w', 'tsc', 'tsc_adjust', 'tsc_deadline_timer', 'vme', 'vmx', 'vnmi', 'vpid', 'x2apic', 'xsave', 'xsaveopt', 'xtopology', 'xtpr']
l3_cache_size: 10485760
l2_cache_size: 262144
l1_data_cache_size: 32768
l1_instruction_cache_size: 32768
```

Appendix Figure 2 –CPU used in the HPC



Appendix Figure 3 –Code Flowchart. Utilized code from "NVIDIA Generative AI with Diffusion Models"[1]