US Patent & Trademark Office
Patent Public Search | Text View

United States Patent Application Publication 20250259277
Kind Code A1
Publication Date August 14, 2025
Inventor(s) Chan; Eric Ryan Wong et al.

GENERAL DATA PRIOR BASED ON LANGEVIN DIFFUSION

Abstract

Embodiments of the present disclosure relate to a general image prior based on Langevin diffusion. An original representation (image, 3D model, audio) is optimized to resemble a data distribution learned by a trained diffusion model. The original representation may be incomplete and is completed by the optimization process. In an embodiment, the diffusion model may be trained to use an additional conditioning input, such as a text prompt. The diffusion model receives a noisy latent as input and generates a denoised output, such as an image. Generally, the diffusion model samples the learned data distribution to produce the output. The conditioning input provides additional constraint that causes the output to be “nudged” towards the learned data distribution. An example synthesis problem is to generate a panorama image that is much larger compared with images used to train the diffusion model.

Inventors: **Chan; Eric Ryan Wong (Alameda, CA), Karras; Tero Tapani (Helsinki, FI), Aittala; Miika Samuli (Helsinki, FI), Aila; Timo Oskari (Tuusula, FI), Laine; Samuli Matias (Vantaa, FI), Chan; Matthew Aaron Wong (Los Altos, CA), De Mello; Shalini (San Francisco, CA), Nagano; Koki (Playa Vista, CA)**

Applicant: **NVIDIA Corporation** (Santa Clara, CA)

Family ID: **96661069**

Appl. No.: **18/884344**

Filed: **September 13, 2024**

Related U.S. Application Data

us-provisional-application US 63552613 20240212

Publication Classification

Int. Cl.: **G06T5/70** (20240101); **G06T5/60** (20240101); **G06T17/20** (20060101)

U.S. Cl.:

CPC **G06T5/70** (20240101); **G06T5/60** (20240101); **G06T17/20** (20130101);

Background/Summary

CLAIM OF PRIORITY [0001] This application claims the benefit of U.S. Provisional Application No. 63/552,613 (Attorney Docket No. 514711) titled “General Image Prior Based on Langevin Diffusion,” filed Feb. 12, 2024, the entire contents of which is incorporated herein by reference.

BACKGROUND

[0002] In a typical generative use, a diffusion model allows the user to draw samples from an estimated data distribution. In the context of images, for example, such a model can generate novel images that look “realistic”, assuming that it was trained on a sufficiently large dataset of real images. Moreover, the synthesis can be conditioned by, say, a text prompt or a reference image, via various methods. The diffusion model itself may accept a conditioning input, in which case the model has to be trained with such input in mind. Alternatively, the diffusion model itself may be agnostic to conditioning, and the desired conditions are imposed post-hoc when sampling the model. The latter approach is more convenient but produces much lower quality images. Furthermore, it may be desirable to utilize a diffusion model trained for one domain (e.g., 2D images) to generate samples in another domain (e.g., 3D objects). There is a need for addressing these issues and/or other issues associated with the prior art.

SUMMARY

[0003] Embodiments of the present disclosure relate to a general image prior based on Langevin diffusion. Systems and methods are disclosed for optimizing an original representation (image, 3D model, audio) to resemble a data distribution learned by a trained diffusion model. The original representation may be incomplete and is completed by the optimization process. Alternatively, the original representation may correspond to an arbitrary initialization (e.g., a solid gray 2D image, a 3D sphere) and is turned into a realistic sample (e.g., a 2D image or a 3D model of a car) by the optimization process. In an embodiment, the diffusion model may be trained to use an additional conditioning input, such as a text prompt. The diffusion model receives a noisy latent as input and generates a denoised output, such as an image. Generally, the diffusion model samples the learned data distribution to produce the output. A generalized method sampling the diffusion model is presented, where the original representation is “nudged” towards the learned data distribution and, at the same time, optionally optimized to satisfy a given set of external constraints. The external constraints might include, for example, a requirement that a certain part of the generated image matches a given reference image, or that the generated 3D object looks like a given reference photo when viewed from a certain angle.

[0004] An example synthesis problem is to generate a panorama image that is much larger compared with images used to train the diffusion model. If the diffusion model was trained with realistic images, this encourages that any crop of the panorama image looks realistic. One or more text prompts may provide additional constraints, such as the desired content of different parts of the panorama. Another example synthesis problem is to generate a 3D object using a diffusion model trained to generate images. In contrast to conventional systems, such as those described above, the trained diffusion model is enhanced to apply an additional constraint for synthesizing an output based on the learned data distribution.

[0005] In an embodiment, the method includes incorporating neutral noise into a rendered original representation to produce a latent representation, applying a Langevin diffusion operation to the latent representation to modify the neutral noise based on a desired data distribution, producing a modified latent, extracting a difference between a denoised latent produced by denoising the latent representation and a denoised modified latent produced by denoising the modified latent representation, and updating the original representation based on the difference. In an embodiment, the rendered original representation, the latent representation, and the modified latent comprise x.sub.0, x.sub.σ, and x.sub.σ′, respectively. In an embodiment, the difference comprises D′(x′.sub.σ; σ)−D′(x.sub.σ; σ), where D′ is an auxiliary denoiser.

Description

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] The present systems and methods for a general image prior based on Langevin diffusion are described in detail below with reference to the attached drawing figures, wherein:

[0007] FIG. **1**A illustrates a block diagram of an example generalized synthesis system suitable for use in implementing some embodiments of the present disclosure.

[0008] FIG. **1**B illustrates a block diagram of an example representation update unit suitable for use in implementing some embodiments of the present disclosure.

[0009] FIG. **2**A illustrates a block diagram of an example Langevin unit suitable for use in implementing some embodiments of the present disclosure.

[0010] FIG. **2**B illustrates a block diagram of an example target rendered representation generation unit suitable for use in implementing some embodiments of the present disclosure.

[0011] FIG. **3** illustrates a flowchart of a method for synthesizing an enhanced representation suitable for use in implementing some embodiments of the present disclosure.

[0012] FIG. **4** illustrates an example parallel processing unit suitable for use in implementing some embodiments of the present disclosure.

[0013] FIG. **5**A is a conceptual diagram of a processing system implemented using the PPU of FIG. **4**, suitable for use in implementing some embodiments of the present disclosure.

[0014] FIG. **5**B illustrates an exemplary system in which the various architecture and/or functionality of the various previous embodiments may be implemented.

[0015] FIG. **5**C illustrates components of an exemplary system that can be used to train and utilize machine learning, in at least one embodiment.

[0016] FIG. **6**A is a conceptual diagram of a graphics processing pipeline implemented by the PPU of FIG. **4** suitable for use in implementing some embodiments of the present disclosure.

[0017] FIG. **6**B illustrates an exemplary streaming system suitable for use in implementing some embodiments of the present disclosure.

DETAILED DESCRIPTION

[0018] Systems and methods are disclosed related to using a general image prior based on Langevin diffusion to optimize an original representation (image, 3D model, audio). The original representation is optimized to resemble a general image prior defined as a data distribution learned by a trained diffusion model. The original representation may be incomplete and is completed by the optimization process. In an embodiment, the diffusion model may be trained to use an additional conditioning input, such as a text prompt. In another embodiment, the original representation may correspond to a different domain (e.g., 3D objects) than what the diffusion model was trained for (e.g., 2D images).

[0019] A trained diffusion model learns a bijective mapping between images and latents, where a latent is a noisy image x at a noise level σ. The latent at noise level σ=0 is the same as the image. Intuitively, the likelihood of the latent is linked to the likelihood of the resulting image. High likelihood latents (i.e., latents that look like a realistic image corrupted with Gaussian white noise) should produce reasonable images, while low likelihood latents (e.g., images with artifacts, incorrect noise profile, significantly skewed standard deviation, etc.) should produce poor-quality images. Images may be improved by improving the likelihood of a latent that produces the image.

[0020] In an embodiment, the trained diffusion model receives a noisy latent as input and generates a denoised output, such as an image. Generally, the diffusion model samples the learned data distribution to produce the output. A generalized method sampling the diffusion model is presented, where the original representation is “nudged” towards the learned data distribution and, at the same time, optionally optimized to satisfy a given set of external constraints. An example synthesis problem is to generate a panorama image that is much larger compared with images used to train the diffusion model. If the diffusion model was trained with realistic images, this encourages that any crop of the panorama image looks realistic. One or more text prompts may provide additional constraints, such as the desired content of different parts of the panorama. Another example synthesis problem is to generate a 3D object using a diffusion model trained to generate images.

[0021] More illustrative information will now be set forth regarding various optional architectures and features with which the foregoing framework may be implemented, per the desires of the user. It should be strongly noted that the following information is set forth for illustrative purposes and should not be construed as limiting in any manner. Any of the following features may be optionally incorporated with or without the exclusion of other features described.

[0022] The theory of diffusion models is rich and complex and is not described in detail except for the parts that are absolutely necessary for understanding the invention. For background on the topic, see Karras et al. Elucidating the design space of diffusion-based generative models. In *proc. NeurIPS*, 2022 which is incorporated by reference.

[0023] Following the notation of Karras et al., the stochastic differential equation (SDE) of interest can be formulated as an equation (1)

$[00001]\mathrm{dx}_{\pm} = -\dot{\ }(t)\ \ (t)\nabla_x \log p(x;\ \ (t))\mathrm{dt} \pm\ \ (t)\ \ (t)^2 \nabla_x \log p(x;\ \ (t))\mathrm{dt} + \sqrt{2\ \ (t)}\ \ (t)d\ \ _t,$

where x is the image being sampled, σ is the noise schedule parameterized by time t, {dot over (σ)}(t) denotes the time derivative of σ, β(t)determines the stochasticity schedule, ω.sub.t is the standard Wiener process, and ∇.sub.x log p(x;σ(t))dt is the score function of p(.Math.), that is typically estimated using a neural network.

[0024] Two SDEs are considered: a forward SDE that describes the process of increasing the amount of noise in an image, and a reverse SDE that describes the process of decreasing it. As seen in the generalized form of equation (1), the SDE is composed of two terms: a probability flow ordinary differential equation (ODE), the first term −{dot over (σ)}(t)σ(t)∇.sub.x log p(x;σ(t))dt in equation (1), and a Langevin diffusion SDE, the second and third terms β(t)σ(t).sup.2∇.sub.x log p(x;σ(t))dt and √{square root over (2β(t))}σ(t)dω.sub.t, respectively, in equation (1). The probability flow ODE moves the sample between noise levels, while the Langevin diffusion SDE adds stochasticity and maintains the same noise level. β(t) is a factor that controls how much stochasticity is included in the diffusion process and is a factor that may be chosen. With the choice of β(t)=0, the Langevin diffusion terms disappear and only the probability flow ODE remains—this would correspond to deterministic sampling.

[0025] At the other extreme, assume a goal is to maintain the same, constant noise level, i.e., {dot over (σ)}(t)=0. The probability flow ODE disappears, and only the two Langevin diffusion terms remain. The Langevin diffusion SDE will drive a sample towards the desired marginal distribution. In other words, given an arbitrary initialization of x.sub.σ and given enough time, the Langevin diffusion SDE will push the latent towards the correct distribution. In theory, no matter where the process starts, following the Langevin diffusion SDE will improve the latent.

[0026] Following the Langevin diffusion SDE in practice means numerically simulating the SDE with discrete steps using an SDE solver. Just as the solution of an ODE can be approximated with Euler's method, the solution of an SDE can be approximated with Euler-Maruyama or related (see Karras et al) SDE solvers. A discrete SDE solver applied to the Langevin diffusion SDE will repeatedly inject random noise into the latent and remove the same amount of noise from the latent, “churning” the latent. The equations below show the Euler-Maruyama steps for a general SDE of the form dX.sub.t=a(X.sub.t,t)dt+b(X.sub.t,t)dWt.

[00002]
$$x'_{t+1} = x_t + a(x_t, t) \quad t + b(x_t, t) \quad W_t \qquad \text{Eq. (2)}$$
$$x_{t+1} \leftarrow x'_{t+1}$$

[0027] The stochastic “churn” has been shown to be beneficial for diffusion sampling. Most diffusion models use some level of stochasticity when creating images because the implicit Langevin diffusion helps correct errors made earlier in sampling. When additional constraints are imposed that further corrupt latents, as in solving inverse problems, it has been shown that stochasticity can be even more beneficial. For example, many iterations of churn may be used between each step of denoising to help correct artifacts when inpainting.

[0028] FIG. **1**A illustrates a block diagram of an example generalized synthesis system **100** suitable for use in implementing some embodiments of the present disclosure. The generalized synthesis system **100** includes a representation update unit **110**, a rendering function **120**, an inversion unit **130**, a Langevin unit **140**, and a target rendered representation generation unit **150**. The Langevin unit **140** includes a trained diffusion model and receives a latent and, in an embodiment, also receives a conditioning input. The Langevin unit **140** is trained to generate a modified latent that is used to synthesize an output representation, such as a crop of a panoramic image based on an original rendered representation, such as an image.

[0029] It should be understood that this and other arrangements described herein are set forth only as examples. Other arrangements and elements (e.g., machines, interfaces, functions, orders, groupings of functions, etc.) may be used in addition to or instead of those shown, and some elements may be omitted altogether. Further, many of the elements described herein are functional entities that may be implemented as discrete or distributed components or in conjunction with other components, and in any suitable combination and location. Various functions described herein as being performed by entities may be carried out by hardware, firmware, and/or software. For instance, various functions may be carried out by a processor executing instructions stored in memory. Furthermore, persons of ordinary skill in the art will understand that any system that performs the operations of the generalized synthesis system **100** is within the scope and spirit of embodiments of the present disclosure.

[0030] In order to “nudge” an image towards the data distribution, a basic strategy is to improve the likelihood of its latent(s) via Langevin diffusion. This differs from the situation in equation (2) in that the noisy latent x.sub.σ that could be directly updated is not available. Instead, the original representation is rendered and is inverted in order to obtain the noisy latent, and then a change resulting from equation (2) is propagated back to the original representation.

[0031] The input representation update unit **110** simply passes the original representation to the rendering function **120** without modification as the updated representation for a first processing pass. The rendering function **120** processes the updated representation according to rendering parameters, producing a rendered representation (e.g., rendered image of a 3D object, a crop of a 2D panorama image, etc.). Because the diffusion model operates on a latent, considered to be a noisy image, the inversion unit **130** adds noise to an input image x (rendered representation) to reach a noise level σ. The inversion unit **130** incorporates “neutral” noise into the rendered representation, producing a latent. The idea is that the output of the inversion should appear as if it was sampled from the distribution of noisy latents, but at the same time, it should still correspond to the same underlying content as the input image x.sub.σ i.e., incorporating the noise should not inadvertently change the underlying image. In other words, the neutral noise should not overwhelm the underlying content.

[0032] The latent is processed by the Langevin unit **140** to synthesize an output that is a modified version of the latent input. The target rendered representation generation unit **150** processes the modified latent, the latent, and the rendered representation to produce a target rendered representation. The original representation is modified using the target rendered representation.

Several iterations of processing may be performed to complete and/or refine the updated representation, possibly with varying rendering parameters (camera angle, crop rectangle, etc.). After processing is complete, the updated representation is the output.

[0033] In an embodiment, the original representation is not simply stored as a grid of pixels, but may instead be represented by a rendering function render (.Math.). When generating a 3D object using, for example, a neural radiance field (NeRF) as the original representation, latent x.sub.σ is thus the result of rendering an image by the rendering function **120** where a 3D object or scene (original representation) is defined by a NeRF multilayer perceptron (MLP). The resulting rendered representation (image) produced by the rendering function **120** is lifted to noise level σ by the inversion unit **130**. Because x.sub.σ cannot be directly updated, a gradient is backpropagated into the MLP. In this case, to obtain x.sub.σ an image is rendered by an MLP using weights ϕ defining the original representation, and the rendered image is inverted by the inversion unit **130** to obtain a latent x.sub.σ at noise level σ such that the latent reconstructs the rendered image:

[00003] $x \quad = \text{invert}(\text{render}(\quad , \quad), \quad), \quad \text{Eq.}(3)$

where ϕ is the original representation and ψ denotes a set of rendering parameters, such as a camera pose. In this case, x.sub.σ is the result of a complex combination of operations, and x.sub.σ cannot be directly updated. If a step of Euler-Maruyama provides a target latent X′.sub.t+1, in an embodiment, the best option is nudge the parameters (weights) of the MLP, ϕ in order to push x.sub.σ towards x.sub.t+1′. Thus, the second step of equation (2) may not be fulfilled exactly.

[0034] For the simpler task of panorama image synthesis, the panorama image may be represented in pixel space, and then the pixels can be updated in the current crop according to equation (2). The latent x.sub.σ comprises a rendered representation including neutral noise that appears to be drawn from the Gaussian distribution. Conventionally, the latent is “pure noise” from which a diffusion process synthesizes an image. In the context of the following description, the neutral noise is added to the rendered representation to reach a desired noise level.

[0035] FIG. **1**B illustrates a block diagram of an example representation update unit **110** suitable for use in implementing some embodiments of the present disclosure. The representation update unit **110** includes a stored representation **105**, a rendering function **115**, a loss function **125**, and a gradient-based optimization algorithm **135**. For the first processing iteration, the stored representation is the original representation **105**. The stored representation **105** is updated prior to each subsequent processing iteration according to representation updates computed by the gradient-based optimization algorithm **135**. The updated representation is output by the representation update unit **110** to the rendering function **120**.

[0036] Within the representation update unit **110** the rendering function **115** renders the updated representation (original representation for the first iteration). In an embodiment, the rendering function **115** is the same as the rendering function **120**. In another embodiment, the rendering function **120** is different and is not necessarily differentiable. In an embodiment, the rendering function **115** is lower resolution or lower fidelity compared with the rendering function **120**. In an embodiment, the rendering function **115** is configured to render at varying resolutions or fidelity for at least some of the processing iterations to generate a rendered representation (e.g., image).

[0037] A loss function **125** evaluates an objective function for the target rendered representation and the rendered representation of the updated representation to compute a loss. Based on the loss, a gradient-based optimization algorithm **135** computes representation updates to modify the original representation. In an embodiment, the gradient-based optimization algorithm **135** performs the optimization using an algorithm such as stochastic gradient descent (SGD), Adam, and the like. The representation updates are computed and applied to the original representation, for example to synthesize a panorama image or video images as the output. The operations performed by **115**, **125**, and **135** may be repeated multiple times for each iteration through the generalized synthesis system **100** shown in FIG. **1**A.

[0038] FIG. **2**A illustrates a block diagram of an example Langevin unit **140** suitable for use in implementing some embodiments of the present disclosure. The Langevin unit **140** performs an Euler-Maruyama churn step (see Algorithm 1 in TABLE 1) to apply Langevin diffusion, to the latent x.sub.σ to modify the noise in the latent based on a desired data distribution (learned by a diffusion model denoiser **230** D), producing a modified latent. In an embodiment, the latent and conditioning input are processed by the diffusion model denoiser **230**.

[0039] The diffusion model denoiser **230** outputs denoised data, such as an image, and the noise reduction unit **240** takes a linear combination between the input latent and the denoised data to arrive at data with less noise than in the input latent. The noise injection unit **220** then adds fresh generated noise produced by the noise generator **225** that is sufficient to reach the original noise level, producing the modified latent.

[0040] Algorithm **1** shown in TABLE 1 provides pseudocode for how one iteration of the Langevin diffusion SDE can be simulated to produce a clean image. The “noise reduction” performed by the noise reduction unit **240** corresponds to the sum x.sub.σ+D(x.sub.σ; σ)−x.sub.σ)βΔt of the E-M churn step. The “noise injection” performed by the noise injection unit **220** corresponds to the addition of √{square root over (2β)}σ√{square root over (Δte)} of the E-M (Euler-Maruyama) churn step. The noise reduction and noise injection are both linear operations and are simple computations. However, the constant factors β and ∈ influence the ability to achieve the correct noise levels.

TABLE-US-00001 TABLE 1 Langevin diffusion pseudocode Algorithm 1 Langevin Diffusion on Images Require: ϕ representation weights Ψ rendering parameters D main denoiser model D′ auxiliary denoiser model σ noise level Δt timestep x.sub.0 = render(ϕ, Ψ) custom-character render an image x.sub.σ = argmin.sub.x||D′(x; σ) − x.sub.0||.sub.2.sup.2 custom-character invert the rendering x′.sub.σ = x.sub.σ + (D(x.sub.σ; σ) − x.sub.σ)βΔt + custom-character perform E-M chum step {square root over (2βσ)}{square root over (Δtϵ)} where ϵ ~ custom-character (0, I) {circumflex over (x)}.sub.0 = x.sub.0 + D′(x′.sub.σ; σ) − D′(x.sub.σ; σ) custom-character create a target rendering ϕ ← argmin.sub.ϕ||render(ϕ, Ψ) − {circumflex over (x)}.sub.0||.sub.2.sup.2 custom-character update representation

[0041] In the context of an external optimization loop that enforces some other constraints on ϕ, Algorithm 1 can be additionally executed in each iteration to encourage the resulting x.sub.0 to obey the data distribution of D, e.g., look realistic. At each iteration, an image x.sub.0 is rendered by rendering function **120** from the abstract representation that may be a NeRF, a 2D MLP, a panorama, etc. Then the image is inverted by the inversion unit **130**, lifting the image to a latent at noise level σ. A Langevin “churn” is applied by the Langevin unit **140** taking the Euler-Maruyama step to obtain a churned latent x′, providing an on average a higher-likelihood latent. The target rendered representation generation unit **150** creates a target rendering from the churned latent. Finally, the abstract representation is updated by the representation update unit **110** in order to apply the effect of the churn in image space. Note that D′(x.sub.σ′;σ)−D′(x.sub.σ; σ) in Algorithm **1** is the change in the image due purely to the churn, and the update resulting from the churn is incorporated into the rendering. In practice, the churn is approximated (i.e., the argmin.sub.ϕ operation in the pseudocode) through gradient descent, using several internal iterations performed by the representation update unit **110**.

[0042] FIG. **2**B illustrates a block diagram of an example target rendered representation generation unit **150** suitable for use in implementing some embodiments of the present disclosure. The target rendered representation generation unit **150** includes an auxiliary denoiser **155** that denoises the latent generated by the inversion unit **130** to produce a denoised latent. The target rendered representation generation unit **150** also denoises the modified latent using an auxiliary denoiser **155** to produce a denoised modified latent. The target rendered representation generation unit **150** then extracts the noise difference D′(x′; σ)−D′(x.sub.σ; σ) using the denoised latent and the denoised modified latent.

[0043] The pseudocode in TABLE 1 also illustrates the possibility of using two distinct denoiser models D and D′. The main denoiser model D (diffusion model denoiser **230**) determines the distribution towards which the churn attempts to nudge the latent. The nudging may be, e.g., conditioned using a text prompt as the conditioning input to the Langevin unit **140**, and the quality of the denoiser model generally defines the quality of the obtained samples. The auxiliary denoiser model D′(auxiliary denoisers **155**) is used only to move between noise-free images and noise level σ used in the churn step. Therefore, the auxiliary denoiser model may be computationally cheaper than D without major adverse effects, perhaps unconditioned or a smaller denoiser model than D. It is naturally also possible to use the same denoiser model for both purposes, i.e., set D′=D.

[0044] Finally, the target rendered representation generation unit **150** creates a noise-free target rendered representation {circumflex over (x)}.sub.0, by incorporating (summing) the noise difference into the rendered representation. In other words, during the Euler-Maruyama churn step, the Langevin unit **140** replaces the neutral noise with the noise difference which is based on the Langevin diffusion SDE. The resulting modified latent is more consistent with the data manifold learned by the diffusion model denoiser **230** and therefore, more likely to correspond to a realistic image underneath the noise. Compared with the neutral noise, the Langevin diffusion SDE pushes the latent towards the desired data distribution. The Langevin diffusion SDE adds stochasticity while maintaining the noise level σ.

[0045] The noise-free target rendered representation is then used by the representation update unit **110** to update the original representation. In a simple scenario such as a panorama image, updating the original representation may simply replace part of the original representation (the crop used) with the noise-free target rendered representation {circumflex over (x)}.sub.0. Construction of the target rendered representation involves denoising both the latent input to the Langevin unit **140** before the Euler-Maruyama churn step and the modified latent generated by the Langevin unit **140**. The difference between the denoised results provides a change that is injected into the rendered representation. As shown in FIG. **2**B, the target rendered representation is the input rendered representation plus the change (noise difference).

[0046] The representation update unit **110** of FIG. **1**B may also be used to update a 3D model, where the original representation is a 3D model that is indirectly updated by the Langevin unit **140**. The 3D model is rendered by the differentiable rendering function **115** to produce a rendered representation (image). Gradients computed based on the target rendered representation and rendered representation are backpropagated through the rendering function **115** to update the original representation and update the stored representation **105** (3D model). The conditioning input for generating a realistic 3D model may be a text prompt describing what the rendered object should look like. The rendering parameters comprise different camera poses to provide different viewing angles.

[0047] Returning to the inversion operation performed by the inversion unit **130** of FIG. **1**A, several different strategies may be considered for realizing the image inversion, i.e., the argmin.sub.x operation in the pseudocode shown in TABLE 1. Here, x.sub.0 is defined as the image to be inverted. A naïve approach is to simply add noise, i.e., setting x.sub.σ=x.sub.0+∈, where ∈˜custom-character(0,σ.sup.2I), turns out to be a poor solution because the randomness in e dominates the content in the underlying image x.sub.0. At even small σ, the inverted latent contains little information about x.sub.0, and D′(x.sub.0+∈; σ) will often point towards an entirely different image than x.sub.0.

[0048] A better option for performing the inversion operation is the forward probability flow ODE inversion. This involves taking N steps of the probability flow ODE in the direction of increasing noise, from σ.sub.min to σ.sub.max. Example parameters are N=12, σ.sub.min=0.02, σ.sub.max=1. While forward probability flow ODE inversion sometimes achieves good results, it requires evaluating D′(x.sub.0, σ.sub.min), i.e., extracting noise out of an image that contains none. If the current rendering is not precisely on the manifold of good images, then this can cause divergence.

A technique for making the inversion more robust is to add a small amount of noise to the rendering before running the forward ODE, which makes the first network evaluation D′(x.sub.0+∈; σ.sub.min), where ∈~custom-character(0, σ.sub.minI). However, this does not make the inversion entirely robust, and small values of σ.sub.min must be used for the same reasons as in naïve inversion.

[0049] The inversion technique that provides the best results is backprop inversion. First, x.sub.σ is initialized, x.sub.σ=x.sub.0+∈, where ∈~custom-character(0, σ.sup.2I), identically to the naïve inversion technique. However, argmin.sub.x.sub.σ‖D′(x.sub.σ, σ)−x.sub.0‖.sub.2.sup.2 is then optimized in order to push x.sub.σ towards greater alignment with x.sub.0. In practice, solid results have been achieved with **20** iterations of optimization using Adam. A key difference between backprop inversion and ODE inversion is that while ODE inversion considers the rendered original representation x.sub.σ to be in the distribution of clean images p(x; 0), backprop inversion instead assumes the rendered original representation to be in the distribution of denoised images p(D′(x;σ)). At low σ, there may be little distinction between the two, but at higher σ, backprop inversion will expect x.sub.0 to be a blurry image (since the outputs of D′ at high noise levels are typically blurry), whereas the ODE will expect a sharp image because σ.sub.min≈0. The reason backprop inversion is preferred is this property—at moderately high σ, e.g. σ=3, both D′(x′.sub.σ;σ) and D′(x.sub.0; σ) will be blurry, making the desired update also blurry and therefore easier to apply to the rendered representation compared with a high-frequency update.

[0050] FIG. **3** illustrates a flowchart of a method **300** for synthesizing an enhanced representation suitable for use in implementing some embodiments of the present disclosure. Each block of method **300**, described herein, comprises a computing process that may be performed using any combination of hardware, firmware, and/or software. For instance, various functions may be carried out by a processor executing instructions stored in memory. The method may also be embodied as computer-usable instructions stored on computer storage media. The method may be provided by a standalone application, a service or hosted service (standalone or in combination with another hosted service), or a plug-in to another product, to name a few. In addition, method **300** is described, by way of example, with respect to the generalized synthesis system **100** of FIG. **1**A. However, this method may additionally or alternatively be executed by any one system, or any combination of systems, including, but not limited to, those described herein. Furthermore, persons of ordinary skill in the art will understand that any system that performs method **300** is within the scope and spirit of embodiments of the present disclosure.

[0051] At step **310**, neutral noise is incorporated into a representation x.sub.σ to produce a latent representation, x.sub.σ. In an embodiment, the representation comprises at least one of an image, a video, a panorama image, or audio data.

[0052] At step **320**, a Langevin diffusion operation is applied to the latent representation to modify the neutral noise based on a desired data distribution, producing a modified latent representation x.sub.σ′. In an embodiment, a level of the neutral noise is preserved during the application of the Langevin diffusion operation.

[0053] At step **330**, a difference, D′(x.sub.σ′;σ)−D′(x.sub.σ; σ), between a denoised latent produced by denoising the latent representation and a denoised modified latent produced by denoising the modified latent representation is extracted.

[0054] At step **340**, the representation is updated based on the difference. In an embodiment, the representation x.sub.0 is obtained by rendering an original representation, and the original representation is also updated based on the difference. In an embodiment, the original representation is associated with a first domain (e.g., 3D objects) and the representation x.sub.0 is associated with a second domain (e.g., 2D images). In an embodiment, the original representation comprises at least one of an extended video, a panorama image, a 3D model, or extended audio data. In an embodiment, the original representation is defined by a neural radiance field (NeRF) multilayer perceptron.

[0055] In an embodiment, the updating step **340** comprises modifying weights of the NeRF multilayer perceptron. In an embodiment, the Langevin diffusion operation receives a conditioning input for generating a realistic 3D model as the updated original representation. In an embodiment, the conditioning input comprises a text prompt describing a desired appearance of a rendered image of the 3D model.

[0056] In an embodiment, at least one of steps **310**, **320**, **330**, and **340** is performed on a server or in a data center to generate data, such as an image, and the data is streamed to a user device. In an embodiment, at least one of steps **310**, **320**, **330**, and **340** is performed within a cloud computing environment. In an embodiment, at least one of steps **310**, **320**, **330**, and **340** is performed for training, testing, or certifying a neural network employed in a machine, robot, or autonomous vehicle. In an embodiment, at least one of steps **310**, **320**, **330**, and **340** is performed on a virtual machine comprising a portion of a graphics processing unit. In an embodiment, at least one of steps **310**, **320**, **330**, and **340** is implemented to include advanced error correction, fault-tolerance, and self-healing capabilities.

[0057] Embodiments of the present disclosure relate to a general image prior based on Langevin diffusion. A trained diffusion model denoiser (image denoiser) is supplemented with a conditioning input, such as a text prompt, enabling a synthesis system to produce enhanced representations, such as 3D models, video, and panoramic images. An original representation (image, 3D model, audio) is optimized to resemble a data distribution learned by the trained diffusion model. The original representation may be incomplete and is completed by the optimization process. The diffusion model receives a noisy latent as input and generates a denoised output, such as an image. Generally, the diffusion model samples the learned data distribution to produce the output. The conditioning input provides an additional constraint that causes the output to be “nudged” towards the learned data distribution. An example synthesis problem is to generate a panorama image that is much larger compared with images used to train the diffusion model.

Parallel Processing Architecture

[0058] FIG. **4** illustrates a parallel processing unit (PPU) **400**, in accordance with an embodiment. The PPU **400** may be used to implement the generalized synthesis system **100**. The PPU **400** may be used to implement one or more of the representation update unit **110**, the rendering function **120**, the inversion unit **130**, the Langevin unit **140**, and the target rendered representation generation unit **150** within the generalized synthesis system **100**. In an embodiment, a processor such as the PPU **400** may be configured to implement a neural network model. The neural network model may be implemented as software instructions executed by the processor or, in other embodiments, the processor can include a matrix of hardware elements configured to process a set of inputs (e.g., electrical signals representing values) to generate a set of outputs, which can represent activations of the neural network model. In yet other embodiments, the neural network model can be implemented as a combination of software instructions and processing performed by a matrix of hardware elements. Implementing the neural network model can include determining a set of parameters for the neural network model through, e.g., supervised or unsupervised training of the neural network model as well as, or in the alternative, performing inference using the set of parameters to process novel sets of inputs.

[0059] In an embodiment, the PPU **400** is a multi-threaded processor that is implemented on one or more integrated circuit devices. The PPU **400** is a latency hiding architecture designed to process many threads in parallel. A thread (e.g., a thread of execution) is an instantiation of a set of instructions configured to be executed by the PPU **400**. In an embodiment, the PPU **400** is a graphics processing unit (GPU) configured to implement a graphics rendering pipeline for processing three-dimensional (3D) graphics data in order to generate two-dimensional (2D) image data for display on a display device. In other embodiments, the PPU **400** may be utilized for performing general-purpose computations. While one exemplary parallel processor is provided herein for illustrative purposes, it should be strongly noted that such processor is set forth for illustrative purposes only, and that any processor may be employed to supplement and/or substitute for the same.

[0060] One or more PPUs **400** may be configured to accelerate thousands of High Performance Computing (HPC), data center, cloud computing, and machine learning applications. The PPU **400** may be configured to accelerate numerous deep learning systems and applications for autonomous vehicles, simulation, computational graphics such as ray or path tracing, deep learning, high-accuracy speech, image, and text recognition systems, intelligent video analytics, molecular simulations, drug discovery, disease diagnosis, weather forecasting, big data analytics, astronomy, molecular dynamics simulation, financial modeling, robotics, factory automation, real-time language translation, online search optimizations, and personalized user recommendations, and the like.

[0061] As shown in FIG. **4**, the PPU **400** includes an Input/Output (I/O) unit **405**, a front end unit **415**, a scheduler unit **420**, a work distribution unit **425**, a hub **430**, a crossbar (Xbar) **470**, one or more general processing clusters (GPCs) **450**, and one or more memory partition units **480**. The PPU **400** may be connected to a host processor or other PPUs **400** via one or more high-speed NVLink **410** interconnect. The PPU **400** may be connected to a host processor or other peripheral devices via an interconnect **402**. The PPU **400** may also be connected to a local memory **404** comprising a number of memory devices. In an embodiment, the local memory may comprise a number of dynamic random access memory (DRAM)devices. The DRAM devices may be configured as a high-bandwidth memory (HBM) subsystem, with multiple DRAM dies stacked within each device.

[0062] The NVLink **410** interconnect enables systems to scale and include one or more PPUs **400** combined with one or more CPUs, supports cache coherence between the PPUs **400** and CPUs, and CPU mastering. Data and/or commands may be transmitted by the NVLink **410** through the hub **430** to/from other units of the PPU **400** such as one or more copy engines, a video encoder, a video decoder, a power management unit, etc. (not explicitly shown). The NVLink **410** is described in more detail in conjunction with FIG. **5**B.

[0063] The I/O unit **405** is configured to transmit and receive communications (e.g., commands, data, etc.) from a host processor (not shown) over the interconnect **402**. The I/O unit **405** may communicate with the host processor directly via the interconnect **402** or through one or more intermediate devices such as a memory bridge. In an embodiment, the I/O unit **405** may communicate with one or more other processors, such as one or more the PPUs **400** via the interconnect **402**. In an embodiment, the I/O unit **405** implements a Peripheral Component Interconnect Express (PCIe) interface for communications over a PCIe bus and the interconnect **402** is a PCIe bus. In alternative embodiments, the I/O unit **405** may implement other types of well-known interfaces for communicating with external devices.

[0064] The I/O unit **405** decodes packets received via the interconnect **402**. In an embodiment, the packets represent commands configured to cause the PPU **400** to perform various operations. The I/O unit **405** transmits the decoded commands to various other units of the PPU **400** as the commands may specify. For example, some commands may be transmitted to the front end unit **415**. Other commands may be transmitted to the hub **430** or other units of the PPU **400** such as one or more copy engines, a video encoder, a video decoder, a power management unit, etc. (not explicitly shown). In other words, the I/O unit **405** is configured to route communications between and among the various logical units of the PPU **400**.

[0065] In an embodiment, a program executed by the host processor encodes a command stream in a buffer that provides workloads to the PPU **400** for processing. A workload may comprise several instructions and data to be processed by those instructions. The buffer is a region in a memory that is accessible (e.g., read/write) by both the host processor and the PPU **400**. For example, the I/O unit **405** may be configured to access the buffer in a system memory connected to the interconnect **402** via memory requests transmitted over the interconnect **402**. In an embodiment, the host processor writes the command stream to the buffer and then transmits a pointer to the start of the command stream to the PPU **400**. The front end unit **415** receives pointers to one or more command streams. The front end unit **415** manages the one or more streams, reading commands from the streams and forwarding commands to the various units of the PPU **400**.

[0066] The front end unit **415** is coupled to a scheduler unit **420** that configures the various GPCs **450** to process tasks defined by the one or more streams. The scheduler unit **420** is configured to track state information related to the various tasks managed by the scheduler unit **420**. The state may indicate which GPC **450** a task is assigned to, whether the task is active or inactive, a priority level associated with the task, and so forth. The scheduler unit **420** manages the execution of a plurality of tasks on the one or more GPCs **450**.

[0067] The scheduler unit **420** is coupled to a work distribution unit **425** that is configured to dispatch tasks for execution on the GPCs **450**. The work distribution unit **425** may track a number of scheduled tasks received from the scheduler unit **420**. In an embodiment, the work distribution unit **425** manages a pending task pool and an active task pool for each of the GPCs **450**. As a GPC **450** finishes the execution of a task, that task is evicted from the active task pool for the GPC **450** and one of the other tasks from the pending task pool is selected and scheduled for execution on the GPC **450**. If an active task has been idle on the GPC **450**, such as while waiting for a data dependency to be resolved, then the active task may be evicted from the GPC **450** and returned to the pending task pool while another task in the pending task pool is selected and scheduled for execution on the GPC **450**.

[0068] In an embodiment, a host processor executes a driver kernel that implements an application programming interface (API) that enables one or more applications executing on the host processor to schedule operations for execution on the PPU **400**. In an embodiment, multiple compute applications are simultaneously executed by the PPU **400** and the PPU **400** provides isolation, quality of service (QOS), and independent address spaces for the multiple compute applications. An application may generate instructions (e.g., API calls) that cause the driver kernel to generate one or more tasks for execution by the PPU **400**. The driver kernel outputs tasks to one or more streams being processed by the PPU **400**. Each task may comprise one or more groups of related threads, referred to herein as a warp. In an embodiment, a warp comprises 32 related threads that may be executed in parallel. Cooperating threads may refer to a plurality of threads including instructions to perform the task and that may exchange data through shared memory. The tasks may be allocated to one or more processing units within a GPC **450** and instructions are scheduled for execution by at least one warp.

[0069] The work distribution unit **425** communicates with the one or more GPCs **450** via XBar **470**. The XBar **470** is an interconnect network that couples many of the units of the PPU **400** to other units of the PPU **400**. For example, the XBar **470** may be configured to couple the work distribution unit **425** to a particular GPC **450**. Although not shown explicitly, one or more other units of the PPU **400** may also be connected to the XBar **470** via the hub **430**.

[0070] The tasks are managed by the scheduler unit **420** and dispatched to a GPC **450** by the work distribution unit **425**. The GPC **450** is configured to process the task and generate results. The results may be consumed by other tasks within the GPC **450**, routed to a different GPC **450** via the XBar **470**, or stored in the memory **404**. The results can be written to the memory **404** via the memory partition units **480**, which implement a memory interface for reading and writing data to/from the memory **404**. The results can be transmitted to another PPU **400** or CPU via the NVLink **410**. In an embodiment, the PPU **400** includes a number U of memory partition units **480** that is equal to the number of separate and distinct memory devices of the memory **404** coupled to the PPU **400**. Each GPC **450** may include a memory management unit to provide translation of virtual addresses into physical addresses, memory protection, and arbitration of memory requests. In an embodiment, the memory management unit provides one or more translation lookaside buffers (TLBs) for performing translation of virtual addresses into physical addresses in the memory **404**.

[0071] In an embodiment, the memory partition unit **480** includes a Raster Operations (ROP) unit, a level two(L**2**) cache, and a memory interface that is coupled to the memory **404**. The memory interface may implement 32, 64, 128, 1024-bit data buses, or the like, for high-speed data transfer. The PPU **400** may be connected to up to Y memory devices, such as high bandwidth memory stacks or graphics double-data-rate, version 5, synchronous dynamic random access memory, or other types of persistent storage. In an embodiment, the memory interface implements an HBM**2** memory interface and Y equals half U. In an embodiment, the HBM**2** memory stacks are located on the same physical package as the PPU **400**, providing substantial power and area savings compared with conventional GDDR**5** SDRAM systems. In an embodiment, each HBM**2** stack includes four memory dies and Y equals 4, with each HBM**2** stack including two 128-bit channels per die for a total of 8 channels and a data bus width of 1024 bits.

[0072] In an embodiment, the memory **404** supports Single-Error Correcting Double-Error Detecting (SECDED) Error Correction Code (ECC) to protect data. ECC provides higher reliability for compute applications that are sensitive to data corruption. Reliability is especially important in large-scale cluster computing environments where PPUs **400** process very large datasets and/or run applications for extended periods.

[0073] In an embodiment, the PPU **400** implements a multi-level memory hierarchy. In an embodiment, the memory partition unit **480** supports a unified memory to provide a single unified virtual address space for CPU and PPU **400** memory, enabling data sharing between virtual memory systems. In an embodiment the frequency of accesses by a PPU **400** to memory located on other processors is traced to ensure that memory pages are moved to the physical memory of the PPU **400** that is accessing the pages more frequently. In an embodiment, the NVLink **410** supports address translation services allowing the PPU **400** to directly access a CPU's page tables and providing full access to CPU memory by the PPU **400**.

[0074] In an embodiment, copy engines transfer data between multiple PPUs **400** or between PPUs **400** and CPUs. The copy engines can generate page faults for addresses that are not mapped into the page tables. The memory partition unit **480** can then service the page faults, mapping the addresses into the page table, after which the copy engine can perform the transfer. In a conventional system, memory is pinned (e.g., non-pageable) for multiple copy engine operations between multiple processors, substantially reducing the available memory. With hardware page faulting, addresses can be passed to the copy engines without worrying if the memory pages are resident, and the copy process is transparent.

[0075] Data from the memory **404** or other system memory may be fetched by the memory partition unit **480** and stored in an L**2** cache, which is located on-chip and is shared between the various GPCs **450**. As shown, each memory partition unit **480** includes a portion of the L**2** cache associated with a corresponding memory **404**. Lower level caches may then be implemented in various units within the GPCs **450**. For example, each of the processing units within a GPC **450** may implement a level one (L**1**) cache. The L**1** cache is private memory that is dedicated to a particular processing unit. The L**2** cache is coupled to the memory interface **470** and the XBar **470** and data from the L**2** cache may be fetched and stored in each of the L**1** caches for processing.

[0076] In an embodiment, the processing units within each GPC **450** implement a SIMD (Single-Instruction, Multiple-Data) architecture where each thread in a group of threads (e.g., a warp) is configured to process a different set of data based on the same set of instructions. All threads in the group of threads execute the same instructions. In another embodiment, the processing unit implements a SIMT (Single-Instruction, Multiple Thread) architecture where each thread in a group of threads is configured to process a different set of data based on the same set of instructions, but where individual threads in the group of threads are allowed to diverge during execution. In an embodiment, a program counter, call stack, and execution state is maintained for each warp, enabling concurrency between warps and serial execution within warps when threads within the warp diverge. In another embodiment, a program counter, call stack, and execution state is maintained for each individual thread, enabling equal concurrency between all threads, within and between warps. When execution state is maintained for each individual thread, threads executing the same instructions may be converged and executed in parallel for maximum efficiency.

[0077] Cooperative Groups is a programming model for organizing groups of communicating threads that allows developers to express the granularity at which threads are communicating, enabling the expression of richer, more efficient parallel decompositions. Cooperative launch APIs support synchronization amongst thread blocks for the execution of parallel algorithms. Conventional programming models provide a single, simple construct for synchronizing cooperating threads: a barrier across all threads of a thread block (e.g., the syncthreads ( ) function). However, programmers would often like to define groups of threads at smaller than thread block granularities and synchronize within the defined groups to enable greater performance, design flexibility, and software reuse in the form of collective group-wide function interfaces.

[0078] Cooperative Groups enables programmers to define groups of threads explicitly at sub-block (e.g., as small as a single thread) and multi-block granularities, and to perform collective operations such as synchronization on the threads in a cooperative group. The programming model supports clean composition across software boundaries, so that libraries and utility functions can synchronize safely within their local context without having to make assumptions about convergence. Cooperative Groups primitives enable new patterns of cooperative parallelism, including producer-consumer parallelism, opportunistic parallelism, and global synchronization across an entire grid of thread blocks.

[0079] Each processing unit includes a large number (e.g., 128, etc.) of distinct processing cores (e.g., functional units) that may be fully-pipelined, single-precision, double-precision, and/or mixed precision and include a floating point arithmetic logic unit and an integer arithmetic logic unit. In an embodiment, the floating point arithmetic logic units implement the IEEE 754-2008 standard for floating point arithmetic. In an embodiment, the cores include 64 single-precision (32-bit) floating point cores, 64 integer cores, 32 double-precision (64-bit) floating point cores, and 8 tensor cores.

[0080] Tensor cores configured to perform matrix operations. In particular, the tensor cores are configured to perform deep learning matrix arithmetic, such as GEMM (matrix-matrix multiplication) for convolution operations during neural network training and inferencing. In an embodiment, each tensor core operates on a 4×4 matrix and performs a matrix multiply and accumulate operation $D=A\times B+C$, where A, B, C, and D are 4×4 matrices.

[0081] In an embodiment, the matrix multiply inputs A and B may be integer, fixed-point, or floating point matrices, while the accumulation matrices C and D may be integer, fixed-point, or floating point matrices of equal or higher bitwidths. In an embodiment, tensor cores operate on one, four, or eight bit integer input data with 32-bit integer accumulation. The 8-bit integer matrix multiply requires 1024 operations and results in a full precision product that is then accumulated using 32-bit integer addition with the other intermediate products for a 8×8×16 matrix multiply. In an embodiment, tensor Cores operate on 16-bit floating point input data with 32-bit floating point accumulation. The 16-bit floating point multiply requires 64 operations and results in a full precision product that is then accumulated using 32-bit floating point addition with the other intermediate products for a 4×4×4 matrix multiply. In practice, Tensor Cores are used to perform much larger two-dimensional or higher dimensional matrix operations, built up from these smaller elements. An API, such as CUDA 9 C++ API, exposes specialized matrix load, matrix multiply and accumulate, and matrix store operations to efficiently use Tensor Cores from a CUDA-C++ program. At the CUDA level, the warp-level interface assumes 16×16 size matrices spanning all 32 threads of the warp.

[0082] Each processing unit may also comprise M special function units (SFUs) that perform special functions (e.g., attribute evaluation, reciprocal square root, and the like). In an embodiment, the SFUs may include a tree traversal unit configured to traverse a hierarchical tree data structure.

In an embodiment, the SFUs may include texture unit configured to perform texture map filtering operations. In an embodiment, the texture units are configured to load texture maps (e.g., a 2D array of texels) from the memory **404** and sample the texture maps to produce sampled texture values for use in shader programs executed by the processing unit. In an embodiment, the texture maps are stored in shared memory that may comprise or include an L**1** cache. The texture units implement texture operations such as filtering operations using mip-maps (e.g., texture maps of varying levels of detail). In an embodiment, each processing unit includes two texture units.

[0083] Each processing unit also comprises N load store units (LSUs) that implement load and store operations between the shared memory and the register file. Each processing unit includes an interconnect network that connects each of the cores to the register file and the LSU to the register file, shared memory. In an embodiment, the interconnect network is a crossbar that can be configured to connect any of the cores to any of the registers in the register file and connect the LSUs to the register file and memory locations in shared memory.

[0084] The shared memory is an array of on-chip memory that allows for data storage and communication between the processing units and between threads within a processing unit. In an embodiment, the shared memory comprises 128 KB of storage capacity and is in the path from each of the processing units to the memory partition unit **480**. The shared memory can be used to cache reads and writes. One or more of the shared memory, L**1** cache, L**2** cache, and memory **404** are backing stores.

[0085] Combining data cache and shared memory functionality into a single memory block provides the best overall performance for both types of memory accesses. The capacity is usable as a cache by programs that do not use shared memory. For example, if shared memory is configured to use half of the capacity, texture and load/store operations can use the remaining capacity. Integration within the shared memory enables the shared memory to function as a high-throughput conduit for streaming data while simultaneously providing high-bandwidth and low-latency access to frequently reused data.

[0086] When configured for general purpose parallel computation, a simpler configuration can be used compared with graphics processing. Specifically, fixed function graphics processing units, are bypassed, creating a much simpler programming model. In the general purpose parallel computation configuration, the work distribution unit **425** assigns and distributes blocks of threads directly to the processing units within the GPCs **450**. Threads execute the same program, using a unique thread ID in the calculation to ensure each thread generates unique results, using the processing unit(s) to execute the program and perform calculations, shared memory to communicate between threads, and the LSU to read and write global memory through the shared memory and the memory partition unit **480**. When configured for general purpose parallel computation, the processing units can also write commands that the scheduler unit **420** can use to launch new work on the processing units.

[0087] The PPUs **400** may each include, and/or be configured to perform functions of, one or more processing cores and/or components thereof, such as Tensor Cores (TCs), Tensor Processing Units (TPUs), Pixel Visual Cores (PVCs), Ray Tracing (RT) Cores, Vision Processing Units (VPUs), Graphics Processing Clusters (GPCs), Texture Processing Clusters (TPCs), Streaming Multiprocessors (SMs), Tree Traversal Units (TTUs), Artificial Intelligence Accelerators (AIAs), Deep Learning Accelerators (DLAs), Arithmetic-Logic Units (ALUs), Application-Specific Integrated Circuits (ASICs), Floating Point Units (FPUs), input/output (I/O) elements, peripheral component interconnect (PCI) or peripheral component interconnect express (PCIe) elements, and/or the like.

[0088] The PPU **400** may be included in a desktop computer, a laptop computer, a tablet computer, servers, supercomputers, a smart-phone (e.g., a wireless, hand-held device), personal digital assistant (PDA), a digital camera, a vehicle, a head mounted display, a hand-held electronic device, and the like. In an embodiment, the PPU **400** is embodied on a single semiconductor substrate. In another embodiment, the PPU **400** is included in a system-on-a-chip (SoC) along with one or more other devices such as additional PPUs **400**, the memory **404**, a reduced instruction set computer (RISC) CPU, a memory management unit (MMU), a digital-to-analog converter (DAC), and the like.

[0089] In an embodiment, the PPU **400** may be included on a graphics card that includes one or more memory devices. The graphics card may be configured to interface with a PCIe slot on a motherboard of a desktop computer. In yet another embodiment, the PPU **400** may be an integrated graphics processing unit (iGPU) or parallel processor included in the chipset of the motherboard. In yet another embodiment, the PPU **400** may be realized in reconfigurable hardware. In yet another embodiment, parts of the PPU **400** may be realized in reconfigurable hardware.

Exemplary Computing System

[0090] Systems with multiple GPUs and CPUs are used in a variety of industries as developers expose and leverage more parallelism in applications such as artificial intelligence computing. High-performance GPU-accelerated systems with tens to many thousands of compute nodes are deployed in data centers, research facilities, and supercomputers to solve ever larger problems. As the number of processing devices within the high-performance systems increases, the communication and data transfer mechanisms need to scale to support the increased bandwidth.

[0091] FIG. **5**A is a conceptual diagram of a processing system **500** implemented using the PPU **400** of FIG. **4**, in accordance with an embodiment. The exemplary system **500** may be configured to implement the method **300** shown in FIG. **3**. The processing system **500** includes a CPU **530**, switch **510**, and multiple PPUs **400**, and respective memories **404**.

[0092] The NVLink **410** provides high-speed communication links between each of the PPUs **400**. Although a particular number of NVLink **410** and interconnect **402** connections are illustrated in FIG. **5**B, the number of connections to each PPU **400** and the CPU **530** may vary. The switch **510** interfaces between the interconnect **402** and the CPU **530**. The PPUs **400**, memories **404**, and NVLinks **410** may be situated on a single semiconductor platform to form a parallel processing module **525**. In an embodiment, the switch **510** supports two or more protocols to interface between various different connections and/or links.

[0093] In another embodiment (not shown), the NVLink **410** provides one or more high-speed communication links between each of the PPUs **400** and the CPU **530** and the switch **510** interfaces between the interconnect **402** and each of the PPUs **400**. The PPUs **400**, memories **404**, and interconnect **402** may be situated on a single semiconductor platform to form a parallel processing module **525**. In yet another embodiment (not shown), the interconnect **402** provides one or more communication links between each of the PPUs **400** and the CPU **530** and the switch **510** interfaces between each of the PPUs **400** using the NVLink **410** to provide one or more high-speed communication links between the PPUs **400**. In another embodiment (not shown), the NVLink **410** provides one or more high-speed communication links between the PPUs **400** and the CPU **530** through the switch **510**. In yet another embodiment (not shown), the interconnect **402** provides one or more communication links between each of the PPUs **400** directly. One or more of the NVLink **410** high-speed communication links may be implemented as a physical NVLink interconnect or either an on-chip or on-die interconnect using the same protocol as the NVLink **410**.

[0094] In the context of the present description, a single semiconductor platform may refer to a sole unitary semiconductor-based integrated circuit fabricated on a die or chip. It should be noted that the term single semiconductor platform may also refer to multi-chip modules with increased connectivity which simulate on-chip operation and make substantial improvements over utilizing a conventional bus implementation. Of course, the various circuits or devices may also be situated separately or in various combinations of semiconductor platforms per the desires of the user. Alternately, the parallel processing module **525** may be implemented as a circuit board substrate and each of the PPUs **400** and/or memories **404** may be packaged devices. In an embodiment, the CPU **530**, switch **510**, and the parallel processing module **525** are situated on a single semiconductor platform.

[0095] In an embodiment, the signaling rate of each NVLink **410** is 20 to 25 Gigabits/second and each PPU **400** includes six NVLink **410** interfaces (as shown in FIG. **5**A, five NVLink **410** interfaces are included for each PPU **400**). Each NVLink **410** provides a data transfer rate of 25 Gigabytes/second in each direction, with six links providing 400 Gigabytes/second. The NVLinks **410** can be used exclusively for PPU-to-PPU communication as shown in FIG. **5**A, or some combination of PPU-to-PPU and PPU-to-CPU, when the CPU **530** also includes one or more NVLink **410** interfaces.

[0096] In an embodiment, the NVLink **410** allows direct load/store/atomic access from the CPU **530** to each PPU's **400** memory **404**. In an embodiment, the NVLink **410** supports coherency operations, allowing data read from the memories **404** to be stored in the cache hierarchy of the CPU **530**, reducing cache access latency for the CPU **530**. In an embodiment, the NVLink **410** includes support for Address Translation Services (ATS), allowing the PPU **400** to directly access page tables within the CPU **530**. One or more of the NVLinks **410** may also be configured to operate in a low-power mode.

[0097] FIG. **5**B illustrates an exemplary system **565** in which the various architecture and/or functionality of the various previous embodiments may be implemented. The exemplary system **565** may be configured to implement the method **300** shown in FIG. **3**. As shown, a system **565** is provided including at least one central processing unit **530** that is connected to a communication bus **575**. The communication bus **575** may directly or indirectly couple one or more of the following devices: main memory **540**, network interface **535**, CPU(s) **530**, display device(s) **545**, input device(s) **560**, switch **510**, and parallel processing system **525**. The communication bus **575** may be implemented using any suitable protocol and may represent one or more links or busses, such as an address bus, a data bus, a control bus, or a combination thereof. The communication bus **575** may include one or more bus or link types, such as an industry standard architecture (ISA) bus, an extended industry standard architecture (EISA) bus, a video electronics standards association (VESA) bus, a peripheral component interconnect (PCI) bus, a peripheral component interconnect express (PCIe) bus, HyperTransport, and/or another type of bus or link. In some embodiments, there are direct connections between components. As an example, the CPU(s) **530** may be directly connected to the main memory **540**. Further, the CPU(s) **530** may be directly connected to the parallel processing system **525**. Where there is direct, or point-to-point connection between components, the communication bus **575** may include a PCIe link to carry out the connection. In these examples, a PCI bus need not be included in the system **565**.

[0098] Although the various blocks of FIG. **5**B are shown as connected via the communication bus **575** with lines, this is not intended to be limiting and is for clarity only. For example, in some embodiments, a presentation component, such as display device(s) **545**, may be considered an I/O component, such as input device(s) **560** (e.g., if the display is a touch screen). As another example, the CPU(s) **530** and/or parallel processing system **525** may include memory (e.g., the main memory **540** may be representative of a storage device in addition to the parallel processing system **525**, the CPUs **530**, and/or other components). In other words, the computing device of FIG. **5**B is merely illustrative. Distinction is not made between such categories as “workstation,” “server,” “laptop,” “desktop,” “tablet,” “client device,” “mobile device,” “hand-held device,” “game console,” “electronic control unit (ECU),” “virtual reality system,” and/or other device or system types, as all are contemplated within the scope of the computing device of FIG. **5**B.

[0099] The system **565** also includes a main memory **540**. Control logic (software) and data are stored in the main memory **540** which may take the form of a variety of computer-readable media. The computer-readable media may be any available media that may be accessed by the system **565**. The computer-readable media may include both volatile and nonvolatile media, and removable and non-removable media. By way of example, and not limitation, the computer-readable media may comprise computer-storage media and communication media.

[0100] The computer-storage media may include both volatile and nonvolatile media and/or removable and non-removable media implemented in any method or technology for storage of information such as computer-readable instructions, data structures, program modules, and/or other data types. For example, the main memory **540** may store computer-readable instructions (e.g., that represent a program(s) and/or a program element(s), such as an operating system. Computer-storage media may include, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which may be used to store the desired information and which may be accessed by system **565**. As used herein, computer storage media does not comprise signals per se.

[0101] The computer storage media may embody computer-readable instructions, data structures, program modules, and/or other data types in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term “modulated data signal” may refer to a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, the computer storage media may include wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer-readable media.

[0102] Computer programs, when executed, enable the system **565** to perform various functions. The CPU(s) **530** may be configured to execute at least some of the computer-readable instructions to control one or more components of the system **565** to perform one or more of the methods and/or processes described herein. The CPU(s) **530** may each include one or more cores (e.g., one, two, four, eight, twenty-eight, seventy-two, etc.) that are capable of handling a multitude of software threads simultaneously. The CPU(s) **530** may include any type of processor, and may include different types of processors depending on the type of system **565** implemented (e.g., processors with fewer cores for mobile devices and processors with more cores for servers). For example, depending on the type of system **565**, the processor may be an Advanced RISC Machines (ARM) processor implemented using Reduced Instruction Set Computing (RISC) or an x86 processor implemented using Complex Instruction Set Computing (CISC). The system **565** may include one or more CPUs **530** in addition to one or more microprocessors or supplementary co-processors, such as math co-processors.

[0103] In addition to or alternatively from the CPU(s) **530**, the parallel processing module **525** may be configured to execute at least some of the computer-readable instructions to control one or more components of the system **565** to perform one or more of the methods and/or processes described herein. The parallel processing module **525** may be used by the system **565** to render graphics (e.g., 3D graphics) or perform general purpose computations. For example, the parallel processing module **525** may be used for General-Purpose computing on GPUS (GPGPU). In embodiments, the CPU(s) **530** and/or the parallel processing module **525** may discretely or jointly perform any combination of the methods, processes and/or portions thereof.

[0104] The system **565** also includes input device(s) **560**, the parallel processing system **525**, and display device(s) **545**. The display device(s) **545** may include a display (e.g., a monitor, a touch screen, a television screen, a heads-up-display (HUD), other display types, or a combination thereof), speakers, and/or other presentation components. The display device(s) **545** may receive data from other components (e.g., the parallel processing system **525**, the CPU(s) **530**, etc.), and output the data (e.g., as an image, video, sound, etc.).

[0105] The network interface **535** may enable the system **565** to be logically coupled to other devices including the input devices **560**, the display device(s) **545**, and/or other components, some of which may be built in to(e.g., integrated in) the system **565**. Illustrative input devices **560** include a microphone, mouse, keyboard, joystick, game pad, game controller, satellite dish, scanner, printer, wireless device, etc. The input devices **560** may provide a natural user interface (NUI) that processes air gestures, voice, or other physiological inputs generated by a user. In some instances, inputs may be transmitted to an appropriate network element for further processing. An NUI may implement any combination of speech recognition, stylus recognition, facial recognition, biometric recognition, gesture recognition both on screen and adjacent to the screen, air gestures, head and eye tracking, and touch recognition (as described in more detail below) associated with a display of the system **565**. The system **565** may be include depth cameras, such as stereoscopic camera systems, infrared camera systems, RGB camera systems, touchscreen technology, and combinations of these, for gesture detection and recognition. Additionally, the system **565** may include accelerometers or gyroscopes (e.g., as part of an inertia measurement unit (IMU)) that enable detection of motion. In some examples, the output of the accelerometers or gyroscopes may be used by the system **565** to render immersive augmented reality or virtual reality.

[0106] Further, the system **565** may be coupled to a network (e.g., a telecommunications network, local area network (LAN), wireless network, wide area network (WAN) such as the Internet, peer-to-peer network, cable network, or the like) through a network interface **535** for communication purposes. The system **565** may be included within a distributed network and/or cloud computing environment.

[0107] The network interface **535** may include one or more receivers, transmitters, and/or transceivers that enable the system **565** to communicate with other computing devices via an electronic communication network, included wired and/or wireless communications. The network interface **535** may be implemented as a network interface controller (NIC) that includes one or more data processing units (DPUs) to perform operations such as (for example and without limitation) packet parsing and accelerating network processing and communication. The network interface **535** may include components and functionality to enable communication over any of a number of different networks, such as wireless networks (e.g., Wi-Fi, Z-Wave, Bluetooth, Bluetooth LE, ZigBee, etc.), wired networks (e.g., communicating over Ethernet or InfiniBand), low-power wide-area networks (e.g., LoRaWAN, SigFox, etc.), and/or the Internet.

[0108] The system **565** may also include a secondary storage (not shown). The secondary storage includes, for example, a hard disk drive and/or a removable storage drive, representing a floppy disk drive, a magnetic tape drive, a compact disk drive, digital versatile disk (DVD)drive, recording device, universal serial bus (USB) flash memory. The removable storage drive reads from and/or writes to a removable storage unit in a well-known manner. The system **565** may also include a hard-wired power supply, a battery power supply, or a combination thereof (not shown). The power supply may provide power to the system **565** to enable the components of the system **565** to operate.

[0109] Each of the foregoing modules and/or devices may even be situated on a single semiconductor platform to form the system **565**. Alternately, the various modules may also be situated separately or in various combinations of semiconductor platforms per the desires of the user. While various embodiments have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of a preferred embodiment should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

Example Network Environments

[0110] Network environments suitable for use in implementing embodiments of the disclosure may include one or more client devices, servers, network attached storage (NAS), other backend devices, and/or other device types. The client devices, servers, and/or other device types (e.g., each device) may be implemented on one or more instances of the processing system **500** of FIG. 5A and/or exemplary system **565** of FIG. 5B—e.g., each device may include similar components, features, and/or functionality of the processing system **500** and/or exemplary system **565**.

[0111] Components of a network environment may communicate with each other via a network(s), which may be wired, wireless, or both. The network may include multiple networks, or a network of networks. By way of example, the network may include one or more Wide Area Networks (WANs), one or more Local Area Networks (LANs), one or more public networks such as the Internet and/or a public switched telephone network (PSTN), and/or one or more private networks. Where the network includes a wireless telecommunications network, components such as a base station, a communications tower, or even access points (as well as other components) may provide wireless connectivity.

[0112] Compatible network environments may include one or more peer-to-peer network environments—in which case a server may not be included in a network environment—and one or more client-server network environments—in which case one or more servers may be included in a network environment. In peer-to-peer network environments, functionality described herein with respect to a server(s) may be implemented on any number of client devices.

[0113] In at least one embodiment, a network environment may include one or more cloud-based network environments, a distributed computing environment, a combination thereof, etc. A cloud-based network environment may include a framework layer, a job scheduler, a resource manager, and a distributed file system implemented on one or more of servers, which may include one or more core network servers and/or edge servers. A framework layer may include a framework to support software of a software layer and/or one or more application(s) of an application layer. The software or application(s) may respectively include web-based service software or applications. In embodiments, one or more of the client devices may use the web-based service software or applications (e.g., by accessing the service software and/or applications via one or more application programming interfaces (APIs)). The framework layer may be, but is not limited to, a type of free and open-source software web application framework such as that may use a distributed file system for large-scale data processing (e.g., “big data”).

[0114] A cloud-based network environment may provide cloud computing and/or cloud storage that carries out any combination of computing and/or data storage functions described herein (or one or more portions thereof). Any of these various functions may be distributed over multiple locations from central or core servers (e.g., of one or more data centers that may be distributed across a state, a region, a country, the globe, etc.). If a connection to a user (e.g., a client device) is relatively close to an edge server(s), a core server(s) may designate at least a portion of the functionality to the edge server(s). A cloud-based network environment may be private (e.g., limited to a single organization), may be public (e.g., available to many organizations), and/or a combination thereof (e.g., a hybrid cloud environment).

[0115] The client device(s) may include at least some of the components, features, and functionality of the example processing system **500** of FIG. **5**A and/or exemplary system **565** of FIG. **5**B. By way of example and not limitation, a client device may be embodied as a Personal Computer (PC), a laptop computer, a mobile device, a smartphone, a tablet computer, a smart watch, a wearable computer, a Personal Digital Assistant (PDA), an MP3 player, a virtual reality headset, a Global Positioning System (GPS) or device, a video player, a video camera, a surveillance device or system, a vehicle, a boat, a flying vessel, a virtual machine, a drone, a robot, a handheld communications device, a hospital device, a gaming device or system, an entertainment system, a vehicle computer system, an embedded system controller, a remote control, an appliance, a consumer electronic device, a workstation, an edge device, any combination of these delineated devices, or any other suitable device.

Machine Learning

[0116] Deep neural networks (DNNs)developed on processors, such as the PPU **400** have been used for diverse use cases, from self-driving cars to faster drug development, from automatic image captioning in online image databases to smart real-time language translation in video chat applications. Deep learning is a technique that models the neural learning process of the human brain, continually learning, continually getting smarter, and delivering more accurate results more quickly over time. A child is initially taught by an adult to correctly identify and classify various shapes, eventually being able to identify shapes without any coaching. Similarly, a deep learning or neural learning system needs to be trained in object recognition and classification for it get smarter and more efficient at identifying basic objects, occluded objects, etc., while also assigning context to objects.

[0117] At the simplest level, neurons in the human brain look at various inputs that are received, importance levels are assigned to each of these inputs, and output is passed on to other neurons to act upon. An artificial neuron or perceptron is the most basic model of a neural network. In one example, a perceptron may receive one or more inputs that represent various features of an object that the perceptron is being trained to recognize and classify, and each of these features is assigned a certain weight based on the importance of that feature in defining the shape of an object.

[0118] A deep neural network (DNN) model includes multiple layers of many connected nodes (e.g., perceptrons, Boltzmann machines, radial basis functions, convolutional layers, etc.) that can be trained with enormous amounts of input data to quickly solve complex problems with high accuracy. In one example, a first layer of the DNN model breaks down an input image of an automobile into various sections and looks for basic patterns such as lines and angles. The second layer assembles the lines to look for higher level patterns such as wheels, windshields, and mirrors. The next layer identifies the type of vehicle, and the final few layers generate a label for the input image, identifying the model of a specific automobile brand.

[0119] Once the DNN is trained, the DNN can be deployed and used to identify and classify objects or patterns in a process known as inference. Examples of inference (the process through which a DNN extracts useful information from a given input) include identifying handwritten numbers on checks deposited into ATM machines, identifying images of friends in photos, delivering movie recommendations to over fifty million users, identifying and classifying different types of automobiles, pedestrians, and road hazards in driverless cars, or translating human speech in real-time.

[0120] During training, data flows through the DNN in a forward propagation phase until a prediction is produced that indicates a label corresponding to the input. If the neural network does not correctly label the input, then errors between the correct label and the predicted label are analyzed, and the weights are adjusted for each feature during a backward propagation phase until the DNN correctly labels the input and other inputs in a training dataset. Training complex neural networks requires massive amounts of parallel computing performance, including floating-point multiplications and additions that are supported by the PPU **400**. Inferencing is less compute-intensive than training, being a latency-sensitive process where a trained neural network is applied to new inputs it has not seen before to classify images, detect emotions, identify recommendations, recognize and translate speech, and generally infer new information.

[0121] Neural networks rely heavily on matrix math operations, and complex multi-layered networks require tremendous amounts of floating-point performance and bandwidth for both efficiency and speed. With thousands of processing cores, optimized for matrix math operations, and delivering tens to hundreds of TFLOPS of performance, the PPU **400** is a computing platform capable of delivering performance required for deep neural network-based artificial intelligence and machine learning applications.

[0122] Furthermore, images or 3D models generated applying one or more of the techniques disclosed herein may be used to train, test, or certify DNNs used to recognize objects and environments in the real world. Such images or 3D models may include scenes of roadways, factories, buildings, urban settings, rural settings, humans, animals, and any other physical object or real-world setting. Such images or 3D models may be used to train, test, or certify DNNs that are employed in machines or robots to manipulate, handle, or modify physical objects in the real world. Furthermore, such images or 3D models may be used to train, test, or certify DNNs that are employed in autonomous vehicles to navigate and move the vehicles through the real world. Additionally, images or 3D models generated applying one or more of the techniques disclosed herein may be used to convey information to users of such machines, robots, and vehicles.

[0123] FIG. 5C illustrates components of an exemplary system **555** that can be used to train and utilize machine learning, in accordance with at least one embodiment. As will be discussed, various components can be provided by various combinations of computing devices and resources, or a single computing system, which may be under control of a single entity or multiple entities. Further, aspects may be triggered, initiated, or requested by different entities. In at least one embodiment training of a neural network might be instructed by a provider associated with provider environment **506**, while in at least one embodiment training might be requested by a customer or other user having access to a provider environment through a client device **502** or other such resource. In at least one embodiment, training data (or data to be analyzed by a trained neural network) can be provided by a provider, a user, or a third party content provider **524**. In at least one embodiment, client device **502** may be a vehicle or object that is to be navigated on behalf of a user, for example, which can submit requests and/or receive instructions that assist in navigation of a device.

[0124] In at least one embodiment, requests are able to be submitted across at least one network **504** to be received by a provider environment **506**. In at least one embodiment, a client device may be any appropriate electronic and/or computing devices enabling a user to generate and send such requests, such as, but not limited to, desktop computers, notebook computers, computer servers, smartphones, tablet computers, gaming consoles (portable or otherwise), computer processors, computing logic, and set-top boxes. Network(s) **504** can include any appropriate network for transmitting a request or other such data, as may include Internet, an intranet, an Ethernet, a cellular network, a local area network (LAN), a wide area network (WAN), a personal area network (PAN), an ad hoc network of direct wireless connections among peers, and so on.

[0125] In at least one embodiment, requests can be received at an interface layer **508**, which can forward data to a training and inference manager **532**, in this example. The training and inference manager **532** can be a system or service including hardware and software for managing requests and service corresponding data or content, in at least one embodiment, the training and inference manager **532** can receive a request to train a neural network, and can provide data for a request to a training module **512**. In at least one embodiment, training module **512** can select an appropriate model or neural network to be used, if not specified by the request, and can train a model using relevant training data. In at least one embodiment, training data can be a batch of data stored in a training data repository **514**, received from client device **502**, or obtained from a third party provider **524**. In at least one embodiment, training module **512** can be responsible for training data. A neural network can be any appropriate network, such as a recurrent neural network (RNN) or convolutional neural network (CNN). Once a neural network is trained and successfully evaluated, a trained neural network can be stored in a model repository **516**, for example, that may store different models or networks for users, applications, or services, etc. In at least one embodiment, there may be multiple models for a single application or entity, as may be utilized based on a number of different factors.

[0126] In at least one embodiment, at a subsequent point in time, a request may be received from client device **502** (or another such device) for content (e.g., path determinations) or data that is at least partially determined or impacted by a trained neural network. This request can include, for example, input data to be processed using a neural network to obtain one or more inferences or other output values, classifications, or predictions, or for at least one embodiment, input data can be received by interface layer **508** and directed to inference module **518**, although a different system or service can be used as well. In at least one embodiment, inference module **518** can obtain an appropriate trained network, such as a trained deep neural network (DNN) as discussed herein, from model repository **516** if not already stored locally to inference module **518**. Inference module

518 can provide data as input to a trained network, which can then generate one or more inferences as output. This may include, for example, a classification of an instance of input data. In at least one embodiment, inferences can then be transmitted to client device **502** for display or other communication to a user. In at least one embodiment, context data for a user may also be stored to a user context data repository **522**, which may include data about a user which may be useful as input to a network in generating inferences, or determining data to return to a user after obtaining instances. In at least one embodiment, relevant data, which may include at least some of input or inference data, may also be stored to a local database **534** for processing future requests. In at least one embodiment, a user can use account information or other information to access resources or functionality of a provider environment. In at least one embodiment, if permitted and available, user data may also be collected and used to further train models, in order to provide more accurate inferences for future requests. In at least one embodiment, requests may be received through a user interface to a machine learning application **526** executing on client device **502**, and results displayed through a same interface. A client device can include resources such as a processor **528** and memory **562** for generating a request and processing results or a response, as well as at least one data storage element **552** for storing data for machine learning application **526**.

[0127] In at least one embodiment a processor **528** (or a processor of training module **512** or inference module **518**) will be a central processing unit (CPU). As mentioned, however, resources in such environments can utilize GPUs to process data for at least certain types of requests. With thousands of cores, GPUs, such as PPU **400** are designed to handle substantial parallel workloads and, therefore, have become popular in deep learning for training neural networks and generating predictions. While use of GPUs for offline builds has enabled faster training of larger and more complex models, generating predictions offline implies that either request-time input features cannot be used or predictions must be generated for all permutations of features and stored in a lookup table to serve real-time requests. If a deep learning framework supports a CPU-mode and a model is small and simple enough to perform a feed-forward on a CPU with a reasonable latency, then a service on a CPU instance could host a model. In this case, training can be done offline on a GPU and inference done in real-time on a CPU. If a CPU approach is not viable, then a service can run on a GPU instance. Because GPUs have different performance and cost characteristics than CPUs, however, running a service that offloads a runtime algorithm to a GPU can require it to be designed differently from a CPU based service.

[0128] In at least one embodiment, video data can be provided from client device **502** for enhancement in provider environment **506**. In at least one embodiment, video data can be processed for enhancement on client device **502**. In at least one embodiment, video data may be streamed from a third party content provider **524** and enhanced by third party content provider **524**, provider environment **506**, or client device **502**. In at least one embodiment, video data can be provided from client device **502** for use as training data in provider environment **506**.

[0129] In at least one embodiment, supervised and/or unsupervised training can be performed by the client device **502** and/or the provider environment **506**. In at least one embodiment, a set of training data **514** (e.g., classified or labeled data) is provided as input to function as training data. In at least one embodiment, training data can include instances of at least one type of object for which a neural network is to be trained, as well as information that identifies that type of object. In at least one embodiment, training data might include a set of images that each includes a representation of a type of object, where each image also includes, or is associated with, a label, metadata, classification, or other piece of information identifying a type of object represented in a respective image. Various other types of data may be used as training data as well, as may include text data, audio data, video data, and so on. In at least one embodiment, training data **514** is provided as training input to a training module **512**. In at least one embodiment, training module **512** can be a system or service that includes hardware and software, such as one or more computing devices executing a training application, for training a neural network (or other model or algorithm, etc.). In at least one embodiment, training module **512** receives an instruction or request indicating a type of model to be used for training, in at least one embodiment, a model can be any appropriate statistical model, network, or algorithm useful for such purposes, as may include an artificial neural network, deep learning algorithm, learning classifier, Bayesian network, and so on. In at least one embodiment, training module **512** can select an initial model, or other untrained model, from an appropriate repository **516** and utilize training data **514** to train a model, thereby generating a trained model (e.g., trained deep neural network) that can be used to classify similar types of data, or generate other such inferences. In at least one embodiment where training data is not used, an appropriate initial model can still be selected for training on input data per training module **512**.

[0130] In at least one embodiment, a model can be trained in a number of different ways, as may depend in part upon a type of model selected. In at least one embodiment, a machine learning algorithm can be provided with a set of training data, where a model is a model artifact created by a training process. In at least one embodiment, each instance of training data contains a correct answer (e.g., classification), which can be referred to as a target or target attribute. In at least one embodiment, a learning algorithm finds patterns in training data that map input data attributes to a target, an answer to be predicted, and a machine learning model is output that captures these patterns. In at least one embodiment, a machine learning model can then be used to obtain predictions on new data for which a target is not specified.

[0131] In at least one embodiment, training and inference manager **532** can select from a set of machine learning models including binary classification, multiclass classification, generative, and regression models. In at least one embodiment, a type of model to be used can depend at least in part upon a type of target to be predicted.

Graphics Processing Pipeline

[0132] In an embodiment, the PPU **400** comprises a graphics processing unit (GPU). The PPU **400** is configured to receive commands that specify shader programs for processing graphics data. Graphics data may be defined as a set of primitives such as points, lines, triangles, quads, triangle strips, and the like. Typically, a primitive includes data that specifies a number of vertices for the primitive (e.g., in a model-space coordinate system) as well as attributes associated with each vertex of the primitive. The PPU **400** can be configured to process the graphics primitives to generate a frame buffer (e.g., pixel data for each of the pixels of the display).

[0133] An application writes model data for a scene (e.g., a collection of vertices and attributes) to a memory such as a system memory or memory **404**. The model data defines each of the objects that may be visible on a display. The application then makes an API call to the driver kernel that requests the model data to be rendered and displayed. The driver kernel reads the model data and writes commands to the one or more streams to perform operations to process the model data. The commands may reference different shader programs to be implemented on the processing units within the PPU **400** including one or more of a vertex shader, hull shader, domain shader, geometry shader, and a pixel shader. For example, one or more of the processing units may be configured to execute a vertex shader program that processes a number of vertices defined by the model data. In an embodiment, the different processing units may be configured to execute different shader programs concurrently. For example, a first subset of processing units may be configured to execute a vertex shader program while a second subset of processing units may be configured to execute a pixel shader program. The first subset of processing units processes vertex data to produce processed vertex data and writes the processed vertex data to the L**2** cache and/or the memory **404**. After the processed vertex data is rasterized (e.g., transformed from three-dimensional data into two-dimensional data in screen space) to produce fragment data, the second subset of processing units executes a pixel shader to produce processed fragment data, which is then blended with other processed fragment data and written to the frame buffer in memory **404**. The vertex shader program and pixel shader program may execute concurrently, processing different data from the same scene in a pipelined fashion until all of the model data for the scene has been rendered to the frame buffer. Then, the contents of the frame buffer are transmitted to a display controller for display on a display device.

[0134] FIG. **6**A is a conceptual diagram of a graphics processing pipeline **600** implemented by the PPU **400** of FIG. **4**, in accordance with an embodiment. The graphics processing pipeline **600** is an abstract flow diagram of the processing steps implemented to generate 2D computer-generated images from 3D geometry data. As is well-known, pipeline architectures may perform long latency operations more efficiently by splitting up the operation into a plurality of stages, where the output of each stage is coupled to the input of the next successive stage. Thus, the graphics processing pipeline **600** receives input data **601** that is transmitted from one stage to the next stage of the graphics processing pipeline **600** to generate output data **602**. In an embodiment, the graphics processing pipeline **600** may represent a graphics processing pipeline defined by the OpenGL® API. As an option, the graphics processing pipeline **600** may be implemented in the context of the functionality and architecture of the previous Figures and/or any subsequent Figure(s).

[0135] As shown in FIG. **6**A, the graphics processing pipeline **600** comprises a pipeline architecture that includes a number of stages. The stages include, but are not limited to, a data assembly stage **610**, a vertex shading stage **620**, a primitive assembly stage **630**, a geometry shading stage **640**, a viewport scale, cull, and clip (VSCC) stage **650**, a rasterization stage **660**, a fragment shading stage **670**, and a raster operations stage **680**. In an embodiment, the input data **601** comprises commands that configure the processing units to implement the stages of the graphics processing pipeline **600** and geometric primitives (e.g., points, lines, triangles, quads, triangle strips or fans, etc.) to be processed by the stages. The output data **602** may comprise pixel data (e.g., color data) that is copied into a frame buffer or other type of surface data structure in a memory.

[0136] The data assembly stage **610** receives the input data **601** that specifies vertex data for high-order surfaces, primitives, or the like. The data assembly stage **610** collects the vertex data in a temporary storage or queue, such as by receiving a command from the host processor that includes a pointer to a buffer in memory and reading the vertex data from the buffer. The vertex data is then transmitted to the vertex shading stage **620** for processing.

[0137] The vertex shading stage **620** processes vertex data by performing a set of operations (e.g., a vertex shader or a program) once for each of the vertices. Vertices may be, e.g., specified as a 4-coordinate vector (e.g., <x, y, z, w>) associated with one or more vertex attributes (e.g., color, texture coordinates, surface normal, etc.). The vertex shading stage **620** may manipulate individual vertex attributes such as position, color, texture coordinates, and the like. In other words, the vertex shading stage **620** performs operations on the vertex coordinates or other vertex attributes associated with a vertex. Such operations commonly including lighting operations (e.g., modifying color attributes for a vertex) and transformation operations (e.g., modifying the coordinate space for a vertex). For example, vertices may be specified using coordinates in an object-coordinate space, which are transformed by multiplying the coordinates by a matrix that translates the coordinates from the object-coordinate space into a world space or a normalized-device-coordinate (NCD) space. The vertex shading stage **620** generates transformed vertex data that is transmitted to the primitive assembly stage **630**.

[0138] The primitive assembly stage **630** collects vertices output by the vertex shading stage **620** and groups the vertices into geometric primitives for processing by the geometry shading stage **640**. For example, the primitive assembly stage **630** may be configured to group every three consecutive vertices as a geometric primitive (e.g., a triangle) for transmission to the geometry shading stage **640**. In some embodiments, specific vertices may be reused for consecutive geometric primitives (e.g., two consecutive triangles in a triangle strip may share two vertices). The primitive assembly stage **630** transmits geometric primitives (e.g., a collection of associated vertices) to the geometry shading stage **640**.

[0139] The geometry shading stage **640** processes geometric primitives by performing a set of operations (e.g., a geometry shader or program) on the geometric primitives. Tessellation operations may generate one or more geometric primitives from each geometric primitive. In other words, the geometry shading stage **640** may subdivide each geometric primitive into a finer mesh of two or more geometric primitives for processing by the rest of the graphics processing pipeline **600**. The geometry shading stage **640** transmits geometric primitives to the viewport SCC stage **650**.

[0140] In an embodiment, the graphics processing pipeline **600** may operate within a streaming multiprocessor and the vertex shading stage **620**, the primitive assembly stage **630**, the geometry shading stage **640**, the fragment shading stage **670**, and/or hardware/software associated therewith, may sequentially perform processing operations. Once the sequential processing operations are complete, in an embodiment, the viewport SCC stage **650** may utilize the data. In an embodiment, primitive data processed by one or more of the stages in the graphics processing pipeline **600** may be written to a cache (e.g. L**1** cache, a vertex cache, etc.). In this case, in an embodiment, the viewport SCC stage **650** may access the data in the cache. In an embodiment, the viewport SCC stage **650** and the rasterization stage **660** are implemented as fixed function circuitry.

[0141] The viewport SCC stage **650** performs viewport scaling, culling, and clipping of the geometric primitives. Each surface being rendered to is associated with an abstract camera position. The camera position represents a location of a viewer looking at the scene and defines a viewing frustum that encloses the objects of the scene. The viewing frustum may include a viewing plane, a rear plane, and four clipping planes. Any geometric primitive entirely outside of the viewing frustum may be culled (e.g., discarded) because the geometric primitive will not contribute to the final rendered scene. Any geometric primitive that is partially inside the viewing frustum and partially outside the viewing frustum may be clipped (e.g., transformed into a new geometric primitive that is enclosed within the viewing frustum. Furthermore, geometric primitives may each be scaled based on a depth of the viewing frustum. All potentially visible geometric primitives are then transmitted to the rasterization stage **660**.

[0142] The rasterization stage **660** converts the 3D geometric primitives into 2D fragments (e.g. capable of being utilized for display, etc.). The rasterization stage **660** may be configured to utilize the vertices of the geometric primitives to setup a set of plane equations from which various attributes can be interpolated. The rasterization stage **660** may also compute a coverage mask for a plurality of pixels that indicates whether one or more sample locations for the pixel intercept the geometric primitive. In an embodiment, z-testing may also be performed to determine if the geometric primitive is occluded by other geometric primitives that have already been rasterized. The rasterization stage **660** generates fragment data (e.g., interpolated vertex attributes associated with a particular sample location for each covered pixel) that are transmitted to the fragment shading stage **670**.

[0143] The fragment shading stage **670** processes fragment data by performing a set of operations (e.g., a fragment shader or a program) on each of the fragments. The fragment shading stage **670** may generate pixel data (e.g., color values) for the fragment such as by performing lighting operations or sampling texture maps using interpolated texture coordinates for the fragment. The fragment shading stage **670** generates pixel data that is transmitted to the raster operations stage **680**.

[0144] The raster operations stage **680** may perform various operations on the pixel data such as performing alpha tests, stencil tests, and blending the pixel data with other pixel data corresponding to other fragments associated with the pixel. When the raster operations stage **680** has finished processing the pixel data (e.g., the output data **602**), the pixel data may be written to a render target such as a frame buffer, a color buffer, or the like.

[0145] It will be appreciated that one or more additional stages may be included in the graphics processing pipeline **600** in addition to or in lieu of one or more of the stages described above. Various implementations of the abstract graphics processing pipeline may implement different stages. Furthermore, one or more of the stages described above may be excluded from the graphics processing pipeline in some embodiments (such as the geometry shading stage **640**). Other types of graphics processing pipelines are contemplated as being within the scope of the present disclosure. Furthermore, any of the stages of the graphics processing pipeline **600** may be implemented by one or more dedicated hardware units within a graphics processor such as PPU **400**. Other stages of the graphics processing pipeline **600** may be implemented by programmable hardware units such as the processing unit within the PPU **400**.

[0146] The graphics processing pipeline **600** may be implemented via an application executed by a host processor, such as a CPU. In an embodiment, a device driver may implement an application programming interface (API) that defines various functions that can be utilized by an application in order to generate graphical data for display. The device driver is a software program that includes a plurality of instructions that control the operation of the PPU **400**. The API provides an abstraction for a programmer that lets a programmer utilize specialized graphics hardware, such as the PPU **400**, to generate the graphical data without requiring the programmer to utilize the specific instruction set for the PPU **400**. The application may include an API call that is routed to the device driver for the PPU **400**. The device driver interprets the API call and performs various operations to respond to the API call. In some instances, the device driver may perform operations by executing instructions on the CPU. In other instances, the device driver may perform operations, at least in part, by launching operations on the PPU **400** utilizing an input/output interface between the CPU and the PPU **400**. In an embodiment, the device driver is configured to implement the graphics processing pipeline **600** utilizing the hardware of the PPU **400**.

[0147] Various programs may be executed within the PPU **400** in order to implement the various stages of the graphics processing pipeline **600**. For example, the device driver may launch a kernel on the PPU **400** to perform the vertex shading stage **620** on one processing unit (or multiple processing units). The device driver (or the initial kernel executed by the PPU **400**) may also launch other kernels on the PPU **400** to perform other stages of the graphics processing pipeline **600**, such as the geometry shading stage **640** and the fragment shading stage **670**. In addition, some of the stages of the graphics processing pipeline **600** may be implemented on fixed unit hardware such as a rasterizer or a data assembler implemented within the PPU **400**. It will be appreciated that results from one kernel may be processed by one or more intervening fixed function hardware units before being processed by a subsequent kernel on a processing unit.

[0148] Images generated applying one or more of the techniques disclosed herein may be displayed on a monitor or other display device. In some embodiments, the display device may be coupled directly to the system or processor generating or rendering the images. In other embodiments, the display device may be coupled indirectly to the system or processor such as via a network. Examples of such networks include the Internet, mobile telecommunications networks, a WIFI network, as well as any other wired and/or wireless networking system. When the display device is indirectly coupled, the images generated by the system or processor may be streamed over the network to the display device. Such streaming allows, for example, video games or other applications, which render images, to be executed on a server, a data center, or in a cloud-based computing environment and the rendered images to be transmitted and displayed on one or more user devices (such as a computer, video game console, smartphone, other mobile device, etc.) that are physically separate from the server or data center. Hence, the techniques disclosed herein can be applied to enhance the images that are streamed and to enhance services that stream images such as NVIDIA Geforce Now (GFN), Google Stadia, and the like.

Example Streaming System

[0149] FIG. **6**B is an example system diagram for a streaming system **605**, in accordance with some embodiments of the present disclosure. FIG. **6**B includes server(s) **603** (which may include similar components, features, and/or functionality to the example processing system **500** of FIG. **5**A and/or exemplary system **565** of FIG. **5**B), client device(s) **604** (which may include similar components, features, and/or functionality to the example processing system **500** of FIG. **5**A and/or exemplary system **565** of FIG. **5**B), and network(s) **606** (which may be similar to the network(s)described herein). In some embodiments of the present disclosure, the system **605** may be implemented.

[0150] In an embodiment, the streaming system **605** is a game streaming system and the server(s) **603** are game server(s). In the system **605**, for a game session, the client device(s) **604** may only receive input data in response to inputs to the input device(s) **626**, transmit the input data to the server(s) **603**, receive encoded display data from the server(s) **603**, and display the display data on the display **624**. As such, the more computationally intense computing and processing is offloaded to the server(s) **603** (e.g., rendering—in particular ray or path tracing—for graphical output of the game session is executed by the GPU(s) **615** of the server(s) **603**). In other words, the game session is streamed to the client device(s) **604** from the server(s) **603**, thereby reducing the requirements of the client device(s) **604** for graphics processing and rendering.

[0151] For example, with respect to an instantiation of a game session, a client device **604** may be displaying a frame of the game session on the display **624** based on receiving the display data from the server(s) **603**. The client device **604** may receive an input to one of the input device(s) **626** and generate input data in response. The client device **604** may transmit the input data to the server(s) **603** via the communication interface **621** and over the network(s) **606** (e.g., the Internet), and the server(s) **603** may receive the input data via the communication interface **618**. The CPU(s) **608** may receive the input data, process the input data, and transmit data to the GPU(s) **615** that causes the GPU(s) **615** to generate a rendering of the game session. For example, the input data may be representative of a movement of a character of the user in a game, firing a weapon, reloading, passing a ball, turning a vehicle, etc. The rendering component **612** may render the game session (e.g., representative of the result of the input data) and the render capture component **614** may capture the rendering of the game session as display data (e.g., as image data capturing the rendered frame of the game session). The rendering of the game session may include ray or path-traced lighting and/or shadow effects, computed using one or more parallel processing units-such as GPUs, which may further employ the use of one or more dedicated hardware accelerators or processing cores to perform ray or path-tracing techniques—of the server(s) **603**. The encoder **616** may then encode the display data to generate encoded display data and the encoded display data may be transmitted to the client device **604** over the network(s) **606** via the communication interface **618**. The client device **604** may receive the encoded display data via the communication interface **621** and the decoder **622** may decode the encoded display data to generate the display data. The client device **604** may then display the display data via the display **624**.

[0152] It is noted that the techniques described herein may be embodied in executable instructions stored in a computer readable medium for use by or in connection with a processor-based instruction execution machine, system, apparatus, or device. It will be appreciated by those skilled in the art that, for some embodiments, various types of computer-readable media can be included for storing data. As used herein, a “computer-readable medium” includes one or more of any suitable media for storing the executable instructions of a computer program such that the instruction execution machine, system, apparatus, or device may read (or fetch) the instructions from the computer-readable medium and execute the instructions for carrying out the described embodiments. Suitable storage formats include one or more of an electronic, magnetic, optical, and electromagnetic format. A non-exhaustive list of conventional exemplary computer-readable medium includes: a portable computer diskette; a random-access memory (RAM); a read-only memory (ROM); an erasable programmable read only memory (EPROM); a flash memory device; and optical storage devices, including a portable compact disc (CD), a portable digital video disc (DVD), and the like.

[0153] It should be understood that the arrangement of components illustrated in the attached Figures are for illustrative purposes and that other arrangements are possible. For example, one or

more of the elements described herein may be realized, in whole or in part, as an electronic hardware component. Other elements may be implemented in software, hardware, or a combination of software and hardware. Moreover, some or all of these other elements may be combined, some may be omitted altogether, and additional components may be added while still achieving the functionality described herein. Thus, the subject matter described herein may be embodied in many different variations, and all such variations are contemplated to be within the scope of the claims.
[0154] To facilitate an understanding of the subject matter described herein, many aspects are described in terms of sequences of actions. It will be recognized by those skilled in the art that the various actions may be performed by specialized circuits or circuitry, by program instructions being executed by one or more processors, or by a combination of both. The description herein of any sequence of actions is not intended to imply that the specific order described for performing that sequence must be followed. All methods described herein may be performed in any suitable order unless otherwise indicated herein or otherwise clearly contradicted by context.
[0155] The use of the terms “a” and “an” and “the” and similar references in the context of describing the subject matter (particularly in the context of the following claims) are to be construed to cover both the singular and the plural, unless otherwise indicated herein or clearly contradicted by context. The use of the term “at least one” followed by a list of one or more items (for example, “at least one of A and B”) is to be construed to mean one item selected from the listed items (A or B) or any combination of two or more of the listed items (A and B), unless otherwise indicated herein or clearly contradicted by context. Furthermore, the foregoing description is for the purpose of illustration only, and not for the purpose of limitation, as the scope of protection sought is defined by the claims as set forth hereinafter together with any equivalents thereof. The use of any and all examples, or exemplary language (e.g., “such as”) provided herein, is intended merely to better illustrate the subject matter and does not pose a limitation on the scope of the subject matter unless otherwise claimed. The use of the term “based on” and other like phrases indicating a condition for bringing about a result, both in the claims and in the written description, is not intended to foreclose any other conditions that bring about that result. No language in the specification should be construed as indicating any non-claimed element as essential to the practice of the invention as claimed.

Claims

1. A computer-implemented method, comprising: incorporating neutral noise into a representation to produce a latent representation; applying a Langevin diffusion operation to the latent representation to modify the neutral noise based on a desired data distribution, producing a modified latent representation; extracting a difference between a denoised latent produced by denoising the latent representation and a denoised modified latent produced by denoising the modified latent representation; and updating the representation based on the difference.
2. The computer-implemented method of claim 1, wherein the representation is obtained by rendering an original representation, and further comprising updating the original representation based on the difference.
3. The computer-implemented method of claim 2, wherein the original representation is associated with a first domain and the representation is associated with a second domain.
4. The computer-implemented method of claim 2 wherein the original representation comprises at least one of an extended video, a panorama image, a 3D model, or extended audio data.
5. The computer-implemented method of claim 2, wherein the original representation is defined by a neural radiance field (NeRF) multilayer perceptron.
6. The computer-implemented method of claim 5, wherein the updating comprises modifying weights of the NeRF multilayer perceptron.
7. The computer-implemented method of claim 2, wherein the Langevin diffusion operation

receives a conditioning input for generating a realistic 3D model as the updated original representation.

8. The computer-implemented method of claim 7, wherein the conditioning input comprises a text prompt describing a desired appearance of a rendered image of the 3D model.

9. The computer-implemented method of claim 1, wherein a level of the neutral noise is preserved during the applying.

10. The computer-implemented method of claim 1, wherein the representation comprises at least one of an image, a video, a panorama image, or audio data.

11. The computer-implemented method of claim 1, wherein at least one of the steps of incorporating, applying, extracting, and updating is performed on a server or in a data center to generate data, and the data is streamed to a user device.

12. The computer-implemented method of claim 1, wherein at least one of the steps of incorporating, applying, extracting, and updating is performed within a cloud computing environment.

13. The computer-implemented method of claim 1, wherein at least one of the steps of incorporating, applying, extracting, and updating is performed for training, testing, or certifying a neural network employed in a machine, robot, or autonomous vehicle.

14. The computer-implemented method of claim 1, wherein at least one of the steps of incorporating, applying, extracting, and updating is performed on a virtual machine comprising a portion of a graphics processing unit.

15. The computer-implemented method of claim 1, wherein at least one of the steps of incorporating, applying, extracting, and updating is implemented to include advanced error correction, fault-tolerance, and self-healing capabilities.

16. A synthesis system, comprising: a memory that stores a representation; and a processor that is connected to the memory, wherein the processor is configured to update the representation by: incorporating neutral noise into a rendering of the original representation to produce a latent representation; applying a Langevin diffusion operation to the latent representation to modify the neutral noise based on a desired data distribution, producing a modified latent representation; extracting a difference between a denoised latent produced by denoising the latent representation and a denoised modified latent produced by denoising the modified latent representation; and updating the representation based on the difference.

17. The system of claim 16, wherein the representation is obtained by rendering an original representation, and further comprising updating the original representation based on the difference.

18. The system of claim 17, wherein the original representation is associated with a first domain and the representation x.sub.0 is associated with a second domain.

19. The system of claim 17, wherein the original representation is defined by a neural radiance field (NeRF) multilayer perceptron.

20. A non-transitory computer-readable media storing computer instructions that, when executed by one or more processors, cause the one or more processors to perform the steps of: incorporating neutral noise into a representation to produce a latent representation; applying a Langevin diffusion operation to the latent representation to modify the neutral noise based on a desired data distribution, producing a modified latent representation; extracting a difference between a denoised latent produced by denoising the latent representation and a denoised modified latent produced by denoising the modified latent representation; and updating the representation based on the difference.

21. The non-transitory computer-readable media of claim 20, wherein a level of the neutral noise is preserved during the applying.