CSCI580 Final Report: Optimizing Stable Diffusion

Shane Cranor
Colorado School of Mines
Golden, Colorado, US
shanecranor@mines.edu

Dylan Eck
Colorado School of Mines
Golden, Colorado, US
dylaneck@mines.edu

Abstract

There has been significant research focused on the use of neural networks to generate images based on text prompts. There has been comparatively less research focused on minimizing the memory requirements of these networks. While Stable Diffusion does improve over previous models in terms of memory usage, it still requires ten gigabytes of VRAM to generate images of only 512x512 pixels. Although memory-optimized implementations of Stable Diffusion do exist, robust evaluations of the effects of the optimizations applied are lacking. In this work, we use existing optimized versions of stable diffusion, with additional modifications applied, and evaluate performance by generating images of various sizes using an NVIDIA RTX 2070 GPU with eight gigabytes of VRAM. While the original implementation of Stable Diffusion is unable to even generate 512x512 pixel images on this GPU, we were able to create a modified implementation capable of generating images of up to 1408x1408 pixels with no compromises on generation speed. Given that Stable Diffusion was intended to work with 512x512 pixel images, our results indicate that the modifications applied have the potential to unlock Stable Diffusion on GPUs with even less than eight gigabytes of VRAM.

ACM Reference Format:
Shane Cranor and Dylan Eck. 2022. CSCI580 Final Report: Optimizing Stable Diffusion. In *Proceedings of (CSCI580).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Deep learning image generation models are capable of generating high quality images of arbitrary subjects based on a text description. Stable diffusion is an example of a state of the art image generation model. These models have many uses from creative to commercial, however, generating high resolution images requires a GPU possessing a large quantity of dedicated memory, at least ten gigabytes in the case of Stable Diffusion. [15] This large quantity of memory is necessary to produce high resolution output images. It is possible to reduce memory requirements by reducing output image size. However, higher resolution images generated by Stable Diffusion tend to be more coherent, as seen in Figure 1 and Figure 2. The lower resolution image was produced with the same seed, prompt, and number of samples as the higher resolution image, while appearing over-saturated and more simplistic. Because of this, merely reducing output resolution is not a desirable method for reducing GPU memory usage. The goal for this project is to try to reduce the memory usage of the Stable Diffusion model so it can generate higher resolution images on cheaper consumer GPUs. Decreasing the VRAM requirements for running Stable Diffusion will help democratize the model and make it accessible to people with older and more budget oriented hardware and allow users with enthusiast or data-center GPUs to generate much larger images.

Figure 1. A typical 256x256 image generated by Stable Diffusion with 200 samples with the prompt "A cute cat floating in space high resolution DSLR" and a classifier-free guidance scale of 6.0

CSCI580, November 18, 2022, Final Report

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Figure 2. A typical 704x512 image generated by Stable Diffusion with 20 samples with the prompt "A cute cat floating in space high resolution DSLR" and a classifier-free guidance scale of 6.0

2 Previous Work

The work we put forward in this paper stems directly from the work of Robin Rombach et. al. in their paper *High-Resolution Image Synthesis with Latent Diffusion Models.* [16]. It is in this paper that the latent diffusion model used by Stable Diffusion was first described. Diffusion models already existed and had been used for image synthesis at the time this paper was released. The paper's main contribution was to perform the forward and reverse diffusion processes in a latent space, working on lower resolution compressed version of the input and output images. This compression was learned using deep auto-encoders with the goal of minimizing the amount of information lost when reducing the size of the image description. Likely due to the already decreased model size provided by the use of the latent space encoding, the authors of this paper did not explore any further techniques for reducing the size of the model. However, even with the size reduction obtained through the use of the latent space encoding, the trained Stable Diffusion model still requires at least 10gb of VRAM to generate a 512x512 image. Based on our study of existing literature, It appears that memory optimization of Stable Diffusion and diffusion models more generally have not been explored much in an academic context. This is likely due to a combination of the relative recency of the publication of the Stable Diffusion Paper and, as previously mentioned, that Stable Diffusion itself is already a more memory efficient model than previous diffusion models. We were however, able to find quite a few papers indirectly, but still relatively closely related to our area of research. The papers we reviewed covered a wide range of approaches for optimizing the memory usage and performance of various types of neural networks.

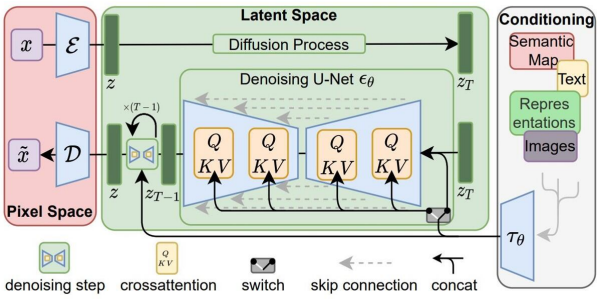

Figure 3. The Stable Diffusion architecture diagram [17].

There were two main types methods of reducing GPU memory usage that we investigated. The first involves compressing the neural network in various ways to reduce the overall amount of memory required to store the full network. The second involved finding ways to reduce the total amount of data that needs to be loaded onto the GPU at any one time.

Papers such as [9] and [6] propose methods for reducing the size of a neural network by carefully removing weights from the network in a way that has a minimal effect on accuracy. These methods both work by evaluating a parameter of each network weight called salience that relates to how much that weight's removal from the network will affect the training error. The two papers calculate salience slightly differently, [9] makes more simplifying assumptions, but the basic idea is the same, calculate the salience of the network weights and remove those weights with the lowest salience. Unfortunately, the method described in [9] requires re-training the network after weights are removed, making its application to Stable Diffusion infeasible. The improved method presented in [6] does not require retraining, but we chose not to investigate it further for reasons that will be discussed later in this section.

Another potential method to reduce network size is quantization. Quantization involves reducing the number of possible values that weights and biases are allowed to have. This results in some weights and or biases having the exact same value as others, allowing for a reduction in the amount of storage used without losing any information. Vector quantization, as discussed in [5] is one such method.

Multiple methods were applied in combination in [5] to reduce to size of deep convolutional neural networks. After applying pruning, trained quantization, and Huffman coding compression, the authors where able to achieve network size reductions between 35 and 45 times without any loss of accuracy.

The techniques for reducing network size and complexity that we explored initially seemed like they would be the most straight forward optimizations for us to implement. However, to implement them, we would need to have a good understanding of, and be able to recreate the low level structure

of the Stable Diffusion neural network. Unfortunately, the data and documentation provided by the creators of Stable Diffusion is geared more toward using the model as opposed to modifying it, so we were unable to gain the level of understanding of Stable Diffusion necessary to apply these kinds of optimizations.

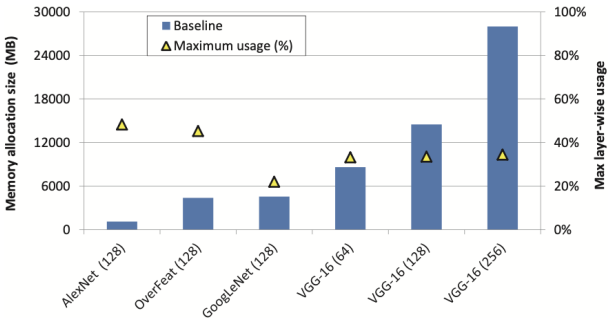

Figure 4. Memory allocation size versus maximum usage when creating a single allocation to store an entire network. [14]

Techniques for reducing the size of memory allocations created on GPUs during neural network training and inference are explored in [14]. This paper focuses specifically on deep convolutional neural networks, but the authors claim that their methods are applicable to any neural network with a multi-layer architecture. In multi-layer networks the GPU can only process a single layer at a time. This is true for both forward passes and back-propagation. However, commonly used neural network frameworks typically load the entire network onto the GPU at the start of training or inference and keep it there for the entire process. As can be seen in 4, this results in a large portion of the data loaded onto the GPU going unused at any given time; anywhere from 53% to 79% of total allocated memory for common convolutional neural networks. When attempting to minimize the size of GPU memory allocations, the authors of this paper focus mostly on feature maps, as they are the most memory expensive component of convolutional neural networks, accounting for upwards of 80% of memory usage for networks such as AlexNet and VGG-16. Feature maps for a given layer in a convolutional neural network are updated during forward passes and are typically kept in memory until the layer is processed again during the following back-propagation. Due to the large size of state of the art convolutional neural networks, it can take a significant amount of time to return to a layer during back-propagation after having processed it in a forward pass, anywhere from 60-1200 ms for AlexNet as an example. As part of their implementation, the authors of this paper remove feature maps from the GPU after they are created during the forward pass, and then re-load then as needed during back-propagation. The implementation also takes care to properly handle more complicated neural network architectures where any given layer's output may be the input to multiple successor layers.

While this method does reduce the size of GPU memory allocations, it also increases the time taken to train and perform inference using a given network. The slowdown is caused by the fact that currently, paging memory between a host machine and a GPU requires transferring pages using the PCIe connection to the GPU, which is not very efficient. PCIe bandwidth utilization when migrating pages between the host system and the GPU is around 80 to 200 MB/s while the maximum available bandwidth is 16 GB/s.

While current methods of paging memory in and out of GPUs are slow [20] investigates potential methods for improving their performance. This paper discusses GPU memory paging, ways to avoid memory over-subscription, and Modifications to GPU architecture. Obviously modifications to GPU architecture go beyond the scope of this project.

The authors of [20] found the GPU memory paging system that they implemented to be 12% faster than manual memory management. While the two papers memory paging is certainly intriguing, we felt that modifying and re-implementing the described memory management systems for use with Stable Diffusion was beyond the scope of what we would be capable of doing within the time-frame of a single semester. Methods for improving neural network performance when running solely on a CPU were explored in [18]. We chose not to pursue these methods since generating images using Stable Diffusion involves a large amount of graphics processing and even older budget graphics cards are much more heavily optimized for this task than CPUs.

In addition to our study of related literature, we also found an examined two existing GitHub repositories that apply various optimizations to Stable Diffusion and claim to allow it to run on graphics cards with much less VRAM than required by the original implementation. The first GitHub repository that we examined was forked from the Stable Diffusion repository by Basu Jindal [7], a masters student at University of California, San Diego. This repository implements two optimizations to reduce the VRAM required to run Stable Diffusion. The first involves splitting the model into chunks that correspond to how it is defined within PyTorch and loading the chunks into the GPU on an as needed basis. The second optimization involves breaking the attention calculation into multiple parts.

The second repository was forked from the Basu Jindal repository by GitHub user consciencia and applies additional optimizations to further reduce VRAM usage. The optimizations applied by this repository include better tensor memory management, the use of flash attention, [3] and performing latent space encoding and decoding on the CPU since those processes do not suffer as much of a performance penalty from being moved to the CPU. The flash attention optimization was applied using the xFormers Python module created

by Facebook Research. This Python module provides efficient, customize python implementations of building blocks of neural network transformer architectures. [13]

Similar ideas to those put forward in the two GitHub repositories were explored in [1]. This paper focuses of creating heavily optimized implementations of primitive structures taken from convolutional neural networks. The concsienca repository applies similar optimization to Stable Diffusion through the use of the xFormers Python Module by implementing memory efficient cross attention.

After finishing our literature review and investigation of the two previously described GitHub Repositories, we decided that the best direction to pursue for this project was to implement a subset of the optimizations found in the two GitHub repositories and perform an in depth analysis of the affects on performance, both with respect to VRAM usage and execution time as well as output image quality.

3 Methodology

Since the resources necessary to train a sophisticated machine learning model such as Stable Diffusion are likely not available to most potential users of the model, we looked at optimizing the use of the trained model rather than the training process; specifically on image generation from a text prompt and not the other features of Stable Diffusion such as image-to-image generation, in-painting, or out-painting. Additionally, our goal is not necessarily to achieve identical or improved performance compared to the original Stable Diffusion. We just want Stable Diffusion to be able to run at a higher resolution on less capable hardware, despite the reduced dedicated memory capacity. This means that side-effects of the memory optimizations, such as a moderately increased image generation time, are acceptable if they allow for the generation of larger images, and as long as the side-effects do not render our implementation unusable or impractical to use.

We set up a reference implementation using the source code provided by the creators of Stable Diffusion (CompVis) on GitHub at [15] as well as the two optimized forks mentioned in the previous work section. The implementations were tested on an 8 core i7-9700k with a NVIDIA ZOTAC AMP 8GB RTX2070 clocked at 1862MHz, running Windows 10. There are three factors that we considered when evaluating the performance of the Stable Diffusion optimizations. First, the largest possible image size that can be generated on our 8GB test GPU. Second, the speed at which the images are generated. Third, the image output quality: if an optimization can generate large images fast, but the image quality is noticeably worse, that optimization may not be a worthwhile trade-off.

For evaluating performance and analysing optimizations, we measure multiple metrics. To profile memory usage and GPU performance we set up NVIDIA Nsight. [10]. We inserted timers into each Stable Diffusion implementation to measure run time and quantify the performance trade-offs that may be a side-effect of memory optimizations. Finally, a foolproof metric will be the range of image output sizes that can be produced without running into out of memory errors. When running with our NVIDIA RTX2070 with 8GB of VRAM, Stable Diffusion throws a CUDA out of memory error when attempting to generate images with any image larger than 448x448.

When running the non-optimized original implementation of Stable Diffusion, we noticed that 100% of the GPU memory is used even when not generating the largest possible image. This would seem to indicated that memory usage is not strongly tied to the size of the image being generated. Due to this fact, measuring memory usage alone may not be the most reliable way to evaluate the improvements provided by the optimizations.

In addition to using the NVIDIA Nsight tools to better analyze the memory usage, we ran Stable Diffusion with a matrix of different image dimensions to make sure our optimizations do not change the quality of the output images and to see if our modifications actually make a meaningful difference to the maximum size of images that can be created. Figure 5 is a matrix representing the time it takes for Stable Diffusion to create a 20 sample image at each resolution from 256x256 to 1024x1024. Each image is generated using the exact same seed, prompt, and other parameters. By comparing this matrix to the matrix produced by the optimized forks, we can determine if the optimizations effect speed and maximum resolution. The non-optimized implementation

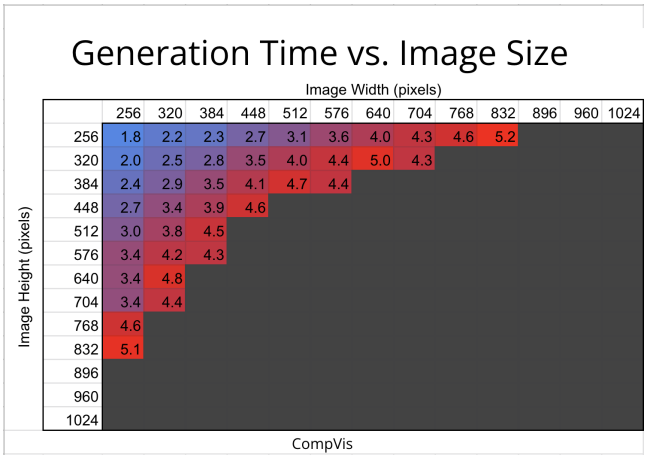

Figure 5. A chart showing the run time in seconds for each possible image resolution in the non-optimized Stable Diffusion run with 20 samples. Gray cells are images dimensions that result in CUDA out of memory errors.

of Stable Diffusion produces a roughly symmetrical matrix where the maximum dimensions for each aspect ratio are

the same when the ratio is inverted, but we test both portrait and landscape resolutions to ensure that the optimizations we apply do not introduce any unintended behaviour or abnormalities affected by the orientation of non-square images. The images in each matrix were generated sequentially with other applications open in the background and GPU temperature is not constant between each run, so some variance in run time is to be expected.

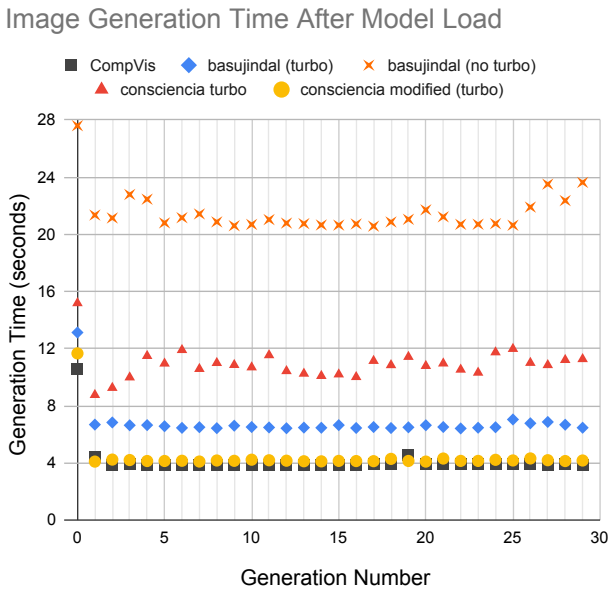

Figure 6. A graph displaying the Nth 448x448 20 sample image after the model is loaded. Notice that generation zero (the first image after model load) consistently takes around six seconds longer, regardless of the optimizations applied

	compvis	basujindal	basujindal (no turbo)	consciencia (turbo)	consciencia modified (turbo)
Standard Deviation	0.156	0.152	0.865	0.732	0.060
Coefficient of Variation	0.040	0.023	0.041	0.068	0.014
(First Render - Median)	6.66 sec	6.61 sec	6.73 sec	4.33 sec	7.52 sec

Figure 7. Statistics from the data in Figure 6

We decided to only load the model once and then run image generations on the preloaded model since we do not take model load time into account when measuring image generation time. However, in each of our matrices, the first 256x256 image generated took twice as long as the 256x320 and 320x256 images. We theorized that perhaps the very first image generated after the model loads takes longer. To test this we generated a series of the exact same 256x256 image and found that the first image took around 12 seconds while the next 6 images took only 6 seconds. Then we tested this hypothesis on a larger 1024x1024 image. The first generation after model load took 50 seconds, while the next 6 images took 43 seconds. This problem could have been introduced by the optimizations that we are testing. However, after testing, we found this to be false, as the unmodified stable diffusion suffers from the same 6 second longer generation time for the first render after model loading as shown in Figure 6. Adding a delay in the form of time.sleep(10) before the first image generation did not yield an increase in performance, so this performance leap is likely the result of some form of caching. To test the variability between runs, we can look at the data in Figure 6. The CompVis, basujindal turbo, and consciencia modified turbo algorithms were fairly consistent within the first 30 runs excluding the first run issue mentioned earlier. The algorithms that had a higher coefficient of variation more heavily utilized the CPU and we hypothesize that the other tasks running on the system and the constant data transfer between CPU and GPU introduced more instability in the run time and thus more variability between generations.

Both basujindal and consciencia forks implemented a command line "`-turbo`" flag. Preliminary testing on the basujindal fork revealed that enabling this flag quadrupled run time at the cost of a slightly lower maximum resolution. On our RTX2070 the maximum resolution without turbo was good enough that it did not make sense to test the non turbo variant. For GPUs with 4GB of VRAM and less disabling the turbo flag could be necessary. While testing the consciencia fork, we noticed that it was taking a long time to save the image, and we realized that this was a consequence of the author moving the decoding stage to the CPU instead of the GPU. The "consciencia modified turbo" algorithm seen in our results is the result of a modification made to the consciencia fork that moves the decoder back to the GPU at the expense of a smaller maximum image size.

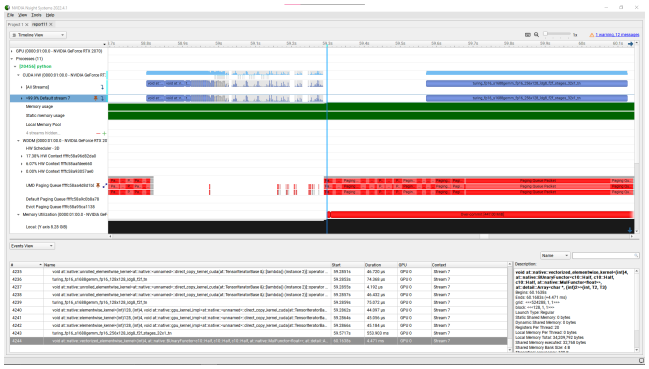

Figure 8. A screenshot of the Nsight Systems 2022.4.1 interface zoomed in on the exact time where the CUDA memory error occurs.

4 Results

We tested the basujindal fork and the consciencia fork with and without several of their optimizations. The charts in

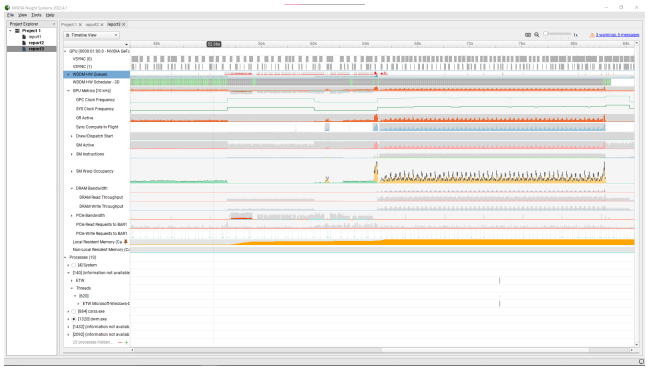

Figure 9. A screenshot of the Nsight Systems 2022.4.1 interface on a successful Stable Diffusion run. The memory usage ramps up to 100% during the image generation.

Figure 10, Figure 11, and Figure 12 show the performance of each memory optimization and can be compared to the unoptimized CompVis repository in Figure 5. Every memory optimization that we tested could generate images from 256x256 to 1024x1024 while CompVis failed at sizes larger than 448x448. Since all the optimizations varied in speed on both small and large images, an easier way of visualizing this data was to multiply the width and height of the generated images and graph that against the generation time as seen in Figure 15. This graph more clearly shows that the modified consciencia fork is capable of creating large images at a two to three times faster than the basujindal fork and the standard consciencia fork. The line of best fit for each fork is a second order polynomial with a good R^2 value. While basujindal turbo performs better at low resolutions than the consciencia turbo, the basujindal fork is faster growing and will take longer at higher resolutions. The modified consciencia fork, although still a second order polynomial, is much slower growing.

Generation Time vs. Image Size

Image Width (pixels)

Image Height (pixels)	256	320	384	448	512	576	640	704	768	832	896	960	1024
256	6.1	6.4	7.0	6.3	6.3	7.0	6.4	6.4	6.4	6.7	6.8	7.9	7.6
320	6.2	7.2	7.8	6.2	6.4	7.0	6.6	6.5	6.4	7.0	7.6	8.0	8.7
384	6.3	6.5	7.5	6.4	6.4	6.8	6.8	7.0	6.8	7.9	8.9	9.7	9.9
448	6.2	6.3	6.5	6.4	6.8	6.9	7.3	7.9	7.4	9.3	10.6	11.1	12.1
512	6.2	6.5	6.4	6.8	6.9	7.3	8.1	8.9	8.6	10.6	12.1	13.7	14.1
576	6.2	6.4	6.6	6.9	7.6	8.4	9.3	10.2	9.6	12.7	14.6	17.0	17.4
640	6.3	6.5	6.8	7.2	8.5	9.3	10.3	11.4	11.2	14.5	17.1	19.1	19.9
704	6.6	6.6	7.0	7.9	9.4	10.2	11.4	13.4	13.1	16.6	19.4	21.5	26.5
768	6.5	7.0	7.4	8.5	10.1	11.2	13.0	14.8	14.7	18.8	25.2	24.1	30.1
832	6.6	7.8	7.9	9.4	11.2	12.7	14.9	16.7	16.7	21.5	28.1	27.1	33.8
896	6.7	9.3	8.5	10.0	12.3	14.0	16.2	18.7	19.0	24.1	32.6	31.0	40.0
960	6.8	10.8	9.2	10.9	14.3	15.5	18.0	20.8	21.2	27.2	33.5	39.6	44.5
1024	7.2	10.0	9.6	11.7	14.1	16.7	21.2	22.9	23.8	30.1	35.1	43.7	43.7

basujindal (turbo)

Figure 10. A chart displaying the run time of the basujindal fork (with –turbo flag) for different image sizes

Generation Time vs. Image Size

Image Width (pixels)

Image Height (pixels)	256	320	384	448	512	576	640	704	768	832	896	960	1024
256	7.1	7.0	6.6	6.8	7.4	7.6	8.0	8.5	8.7	9.5	10.1	11.2	12.0
320	7.1	7.2	7.0	7.5	7.9	8.5	10.3	10.2	10.4	11.1	11.8	12.5	16.2
384	9.2	7.2	7.6	8.2	8.9	9.6	11.4	11.4	12.0	12.9	13.6	14.4	16.9
448	8.5	7.6	8.3	9.2	9.9	10.9	11.8	12.7	14.1	14.5	15.6	16.2	19.3
512	8.5	8.2	8.8	10.0	11.0	12.0	13.3	14.1	15.0	16.4	17.5	18.8	20.4
576	7.8	8.8	9.6	10.8	12.3	13.1	14.4	15.4	16.6	18.1	19.5	20.8	22.4
640	8.0	9.4	10.5	11.9	13.1	14.3	15.8	16.9	18.7	20.5	21.7	23.7	24.9
704	8.7	9.9	11.4	12.8	14.1	15.5	17.0	18.7	20.6	22.6	24.0	25.3	29.1
768	9.5	10.6	12.0	13.6	15.3	16.8	18.7	20.9	22.7	24.2	25.9	27.9	30.7
832	9.6	11.6	12.9	14.6	16.0	18.3	20.2	22.2	24.2	26.0	28.2	30.5	32.5
896	10.8	12.0	13.7	15.8	17.1	19.4	21.9	24.0	26.0	28.4	30.8	32.9	35.4
960	10.9	12.6	14.3	16.5	18.5	21.1	23.4	25.3	27.8	30.4	32.6	36.7	38.1
1024	12.2	13.1	15.3	17.4	19.8	22.6	24.9	27.7	29.9	32.9	35.8	39.0	41.4

consciencia (turbo)

Figure 11. A chart displaying the run time of the consciencia fork (with –turbo flag) for different image sizes

Generation Time vs. Image Size

Image Width (pixels)

Image Height (pixels)	256	320	384	448	512	576	640	704	768	832	896	960	1024
256	4.2	4.3	4.1	4.0	4.0	4.1	4.0	4.1	4.2	4.6	4.8	4.9	4.8
320	4.2	4.8	4.1	4.1	4.3	4.2	4.5	4.5	4.8	5.0	5.2	5.4	5.4
384	4.3	4.9	4.1	4.2	4.2	4.4	4.7	5.0	5.0	5.4	5.5	5.8	5.9
448	4.3	4.2	4.5	4.4	4.5	4.8	5.1	5.4	5.5	5.9	6.2	6.3	6.6
512	4.8	4.1	4.3	4.6	4.7	5.0	5.3	5.6	6.4	6.4	6.6	7.2	7.7
576	4.2	4.2	4.5	4.8	5.1	5.3	5.6	6.1	6.5	7.1	7.7	8.2	8.9
640	4.2	4.4	4.7	5.0	5.3	5.7	6.2	6.6	7.3	8.0	8.5	9.0	10.0
704	4.6	4.5	4.9	5.3	5.6	6.2	6.6	7.2	8.0	8.7	9.3	9.7	11.4
768	4.6	4.7	5.1	5.5	5.9	6.5	7.2	8.1	8.6	9.3	9.7	10.8	12.2
832	4.8	5.0	5.4	5.8	6.4	7.0	7.9	8.6	9.3	10.0	10.9	11.7	13.5
896	5.0	5.1	5.4	6.2	6.6	7.7	8.4	9.2	9.8	10.9	11.8	12.7	15.8
960	4.9	5.4	5.8	6.3	7.1	8.2	9.0	9.7	10.7	11.7	12.8	14.0	16.1
1024	4.9	5.3	5.9	6.6	7.7	8.6	9.2	10.4	11.3	12.6	13.8	15.0	16.5

consciencia modified (turbo)

Figure 12. A chart displaying the run time of the modified consciencia fork (with –turbo flag) for different image sizes

CompVis did not produce enough data due to its limited resolution output, so we can compare the CompVis performance in a version of the previous graph with much of the high pixel count data left out in Figure 16. This graph shows that all optimizations have a moderate performance penalty when it comes to generating smaller images. The modified consciencia turbo optimization actually outperforms CompVis at the largest resolutions that CompVis is capable of rendering on the NVDIA RTX2070. We would need a GPU with more VRAM to determine whether this trend continues at higher resolutions. We also tested how sample size changed the results. In Figure 13 we compare the runtime performance of each optimization against CompVis from 10 samples to 100 samples. The non turbo basujindal run has a very steep linear correlation between samples and generation time compared to CompVis and the optimizations run with the turbo flag. Interestingly, the consciencia turbo

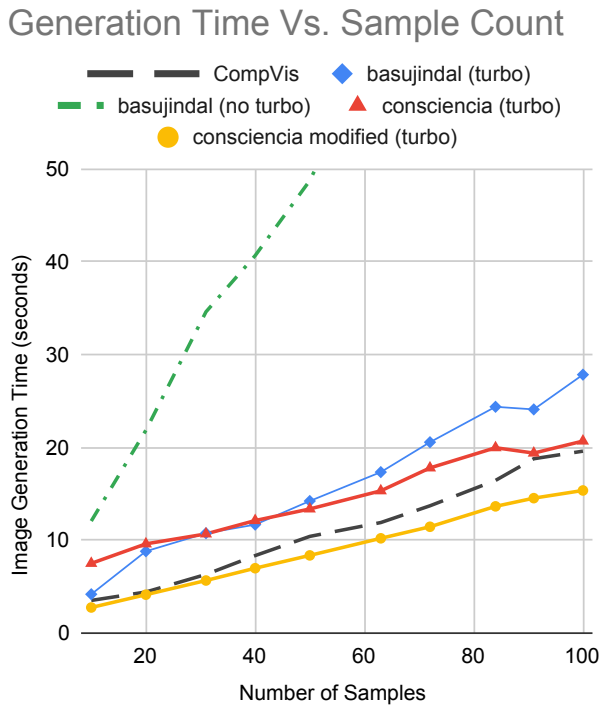

Figure 13. A graph showing how number of samples effects generation time for 448x448 images for each fork

and consciencia modified turbo have about the same slope; however, the non modified version has a slightly higher intercept value. This means that for images with less than 25 samples (at 448x448), basujindal turbo will run faster than consciencia turbo. Not that this matters because consciencia modified turbo blows both of them out of the water, even beating or matching CompVis at all sampling levels.

The modified consciencia has the same slope, but has a larger intercept than the non modified version because the length of the decoding step is not dependent on the number of samples. Our testing shows the length of the decoding step is only dependent on the output resolution, so an image with 10 samples will take just as long to decode as an image with 100 samples, even though the decoding step is on the CPU.

The maximum resolutions of each model can be found in Figure 14. With the consciencia fork producing the largest images at the highest speeds. The basujindal fork still produced impressive results compared to CompVis, although at the sacrifice of speed. All optimizations except the consciencia fork produced exactly the same image for each image in the matrix. The consciencia fork uses xformers which seems to create nondeterministic results. Each generation of the exact same image with the flash attention optimization resulted in a slight variation in the output image. [8]

	Max Square Resolution	Pixels	Generation time
CompVis	440x440	0.2M	4.6 sec
basujindal (turbo)	1152x1152	1.2M	81.5 sec
basujindal (no turbo)	1280x1280	1.6M	227.4 sec
consciencia (turbo)	1984x1984*	3.9M	1061.4 sec
consciencia modified (turbo)	1408x1408	2.0M	40.8 sec

Figure 14. A chart displaying maximum resolution for each model. *given that it took 17 minutes for consciencia (turbo) to produce a 1984x1984 image we did not try to run any larger images, however, it may actually be capable of generating larger images

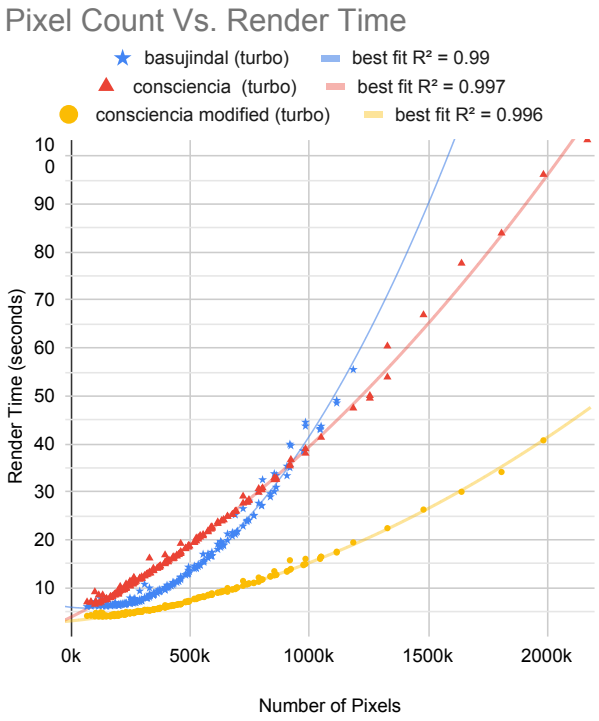

Figure 15. A graph showing the performance of each optimization

5 Difficulties Encountered

One of the initial difficulties we ran into was that Stable diffusion has a somewhat complicated architecture; it took longer than anticipated to gain a good understanding of how it works. Because of this, it was more difficult and took longer than expected to determine a viable method for performing memory optimization. Another difficulty we encountered early on in the project was an apparent lack of research papers closely related to what we were trying to do. Searching IEEE Xplore and Google Scholar yielded papers related to memory optimization for convolutional neural networks [4] and neural networks more generally, [18] [14] but not

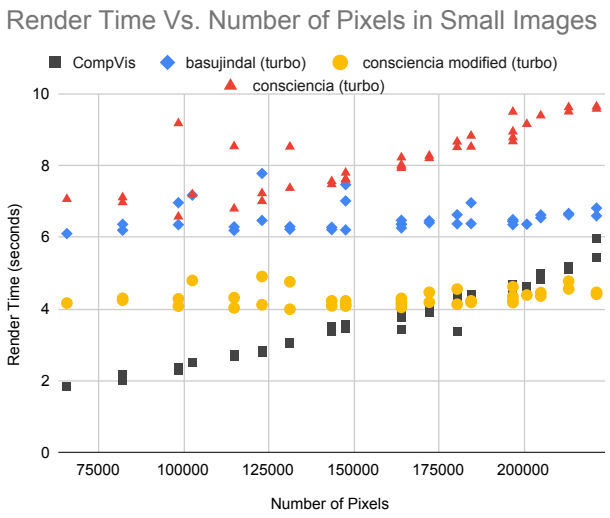

Figure 16. A graph comparing the performance of each optimization to the non-optimized CompVis repository

diffusion models specifically. We also found a lot of papers discussing modifications to GPUs and related hardware to improve machine learning model performance, which, while they did look interesting, did not provide solutions that we could implement on our own in the time frame of a single semester. Despite initial difficulties, we eventually found some papers that seemed promising and we were even able to find several GitHub repositories forked from the original Stable Diffusion repository with some memory optimizations applied to both loading the model as well as performing attention and inference related operations. This research and the recourse we found were helpful in determining how to proceed with the project.

Some difficulties were also encountered when setting up NVIDIA Nsight and the reference installation of Stable Diffusion. We also encountered some difficulties running stable diffusion with the NVIDIA Nsight monitor to analyze memory usage. There are weird bugs that make it difficult to activate the Anaconda3 environment with the python packages required for running Stable Diffusion inside of Nsight. [19] The batch file containing commands to activate the Anaconda environment, or even print the Anaconda version number would fail silently inside Nsight, but run when launched normally in Windows 10. We overcame this difficulty by launching cmd.exe instead of a shell script and then manually running the Conda binary inside of the terminal. According to the documentation and NVIDIA forum, Anaconda should run without difficulty inside of Nsight, however, this was not consistently the case, with the environment not properly activating most of the time despite being on the most up-to date version of Nsight available for download from the NVIDIA website.

Setting up the unmodified Stable Diffusion in a conda environment on windows was trivial; however, running any of the forks that used xformers turned out to be an unexpected challenge. Xformers is relatively new and cutting edge, so it was difficult to find a compiled binary for anything older than the NVDIA RTX30xx series on windows. It also required us to upgrade from Python 3.9 to Python 3.10, which in turn forced us to upgrade many of the pip dependencies in our conda environment.

6 Conclusions

Both optimized Stable Diffusion forks were successful in that they both increased the maximum square resolution for generated images to above 1024x1024: a more than 5x improvement to the original CompVis. Through its use of memory efficient cross attention our modified consciencia fork beats the basujindal fork and the original CompVis in almost every single metric for images 448x448 and larger. The only drawback of using xformers for memory efficient cross attention being the small nondeterministic visual differences in the output. We believe that this visual difference is inconsequential and a tiny price to pay for the 10x increase in maximum resolution at the same speed as or faster than unoptimized Stable Diffusion. The modification that we made to the consciencia fork by keeping the decoder model in the GPU was well worth the increased memory consumption for our 8GB RTX2070. However, the maximum image size will likely be reduced on a lower end GPU with 4GB or 2GB of memory, so experimenting with turning off turbo and or moving the decoder to the CPU could result in more desirable results on lower end hardware. On the other hand, a faster CPU with higher core count could increase the performance of the decoder model on the CPU. This great work from basujindal, consciencia, and other GitHub contributors is lowering the barrier to entry for image generation for people with low end hardware, and increasing the capability of high end hardware. Moving forward, it would be beneficial to investigate using xFormer optimizations with Stable Diffusion model training.

References

[1] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[2] consciencia. 2022. *Optimized Stable Diffusion.* https://github.com/consciencia/stable-diffusion

[3] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. https://doi.org/10.48550/ARXIV.2205.14135

[4] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir D. Bourdev. 2014. Compressing Deep Convolutional Networks using Vector Quantization. *CoRR* abs/1412.6115 (2014). arXiv:1412.6115 http://arxiv.org/abs/1412.6115

[5] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[6] Babak Hassibi and David Stork. 1992. Second order derivatives for network pruning: Optimal Brain Surgeon. In *Advances in Neural Information Processing Systems*, S. Hanson, J. Cowan, and C. Giles (Eds.), Vol. 5. Morgan-Kaufmann. https://proceedings.neurips.cc/paper/1992/file/303ed4c69846ab36c2904d3ba8573050-Paper.pdf

[7] Basu Jindal. 2022. *Optimized Stable Diffusion.* https://github.com/basujindal/stable-diffusion

[8] JustMaier. 2022. *Xformers optimization causes inconsistent generations with the same parameters.* https://github.com/AUTOMATIC1111/stable-diffusion-webui/issues/1999

[9] Yann LeCun, John Denker, and Sara Solla. 1989. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*, D. Touretzky (Ed.), Vol. 2. Morgan-Kaufmann. https://proceedings.neurips.cc/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf

[10] NVIDIA. 2022. *NVIDIA Nsight Graphics.* https://developer.nvidia.com/nsight-graphics

[11] Michela Paganini. 2022. *Pruning Tutorial.* https://pytorch.org/tutorials/intermediate/pruning_tutorial.html

[12] PyTorch. 2022. *Ease-of-Use Quantization for PyTorch With Intel Neural Compressor.* https://pytorch.org/tutorials/recipes/intel_neural_compressor_for_pytorch.html

[13] Facebook Research. 2022. *xFormers - Toolbox to Accelerate Research on Transformers.* https://github.com/facebookresearch/xformers

[14] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783721

[15] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2021. *CompVis/stable-diffusion.* https://github.com/CompVis/stable-diffusion

[16] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2021. High-Resolution Image Synthesis with Latent Diffusion Models. arXiv:2112.10752 [cs.CV]

[17] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. *High-Resolution Image Synthesis with Latent Diffusion Models - (A.K.A. LDM & Stable Diffusion).* https://ommer-lab.com/research/latent-diffusion-models/

[18] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011.*

[19] Yuvalg1987. 2021. *NSight Systems With Anaconda on Windows.* https://forums.developer.nvidia.com/t/nsight-systems-with-anaconda-on-windows/167297

[20] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 345–357. https://doi.org/10.1109/HPCA.2016.7446077

Received 18 November 2022