Diffusion Model Inference on Edge Computing Devices - Nvidia Jetson Nano and Android Smartphone

Akash Haridas
University of Toronto
akashh@cs.toronto.edu

Jazib Ahmad
University of Toronto
jazibahmad@cs.toronto.edu

Sumant Bagri
University of Toronto
sbagri@cs.toronto.edu

Abstract: Despite the popularity of diffusion models for image generation, there remains an unanswered question about their implementation and performance on edge devices. Yet, this is an important question because edge devices promise to decrease the load on the server as well as utilize less expensive, low-power compute. In this project, we answer this question by implementing a Latent Diffusion Model (LDM) on the Nvidia Jetson Nano and an Android smartphone, and comparing its performance on these edge devices against its performance on the Nvidia RTX 3070. Our results show 79.68X and 137.18X inference latency on these devices respectively, as well as 2.93X and 4.76X inference energy consumption. Despite these negative results, we show the promise of better results through various optimizations to the LDM, tested on the RTX 3070. We discuss the challenges with optimizing the LDM on the edge devices. We believe that these challenges are addressed by the recently released Jetson Orin Nano, which would also enable the implementation of even larger models such as stable diffusion. We have made our docker images[1] and evaluation script[2] publicly available to make it easier for anyone to get started with implementing diffusion models on edge devices and evaluating their performance.

Keywords: Diffusion models, edge computing, performance benchmarking, Jetson Nano, Android

1 Introduction

Diffusion models have recently shown state-of-the-art performance in image synthesis and have potential applications in mobile apps and AR/VR. They work by iteratively denoising an initial image to invert a forward noising process, and are capable of producing high-quality images starting from pure noise. Each denoising step involves a forward pass through an $\mathcal{R}^N \rightarrow \mathcal{R}^N$ neural network and a single image generation may involve up to 1000 denoising steps [1]. This iterative process causes diffusion models to be significantly more expensive than previous generative models like GANs, which only require a single forward pass per generated image. Therefore, memory and compute constraints remain major barriers to the practical adoption of diffusion models in resource-constrained applications.

At the same time, Edge Computing promises to decrease the workload on the server as well as decrease the energy cost. In this project, we implement a latent diffusion model (LDM) on memory-restricted edge devices, including the Nvidia Jetson Nano and an Android smartphone. We benchmark their performance metrics including inference latency, memory, power, and energy consumption, and compare it with the performance of the LDM on a Nvidia RTX 3070 laptop GPU. We also implement various optimizations on the RTX 3070, and document the challenges of applying the

[1] https://hub.docker.com/repository/docker/trapdoor20/csc2231/general
[2] https://github.com/SumantBagri/Diffusion-Model-Inference-on-Edge/tree/uldm/eval

Figure 1: Diffusion models generate samples by learning to iteratively remove Gaussian noise starting from pure noise. [1]

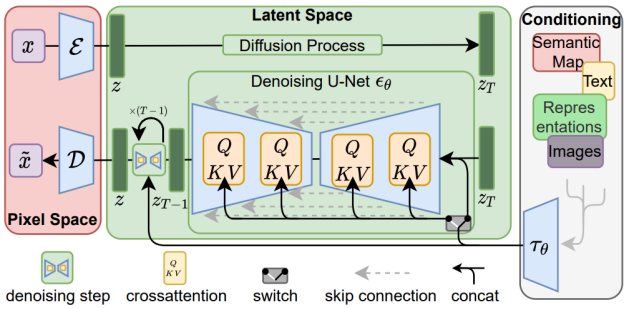


Figure 2: Architecture of the latent diffusion model, which performs denoising updates in the compressed latent space of an encoder network. [3]

same optimizations on our edge devices. Our contributions include open-sourced Docker images that can be utilized to run diffusion models on edge devices, as well as a modular evaluation script that can be used cross-device to measure their performance.

2 Background and Prior Work

In this section, we introduce latent diffusion models (LDMs) and Edge Computing, and the specific edge devices on which we implement LDMs. Here, we also discuss the prior work in benchmarking the performance of deep learning models on these edge devices. Finally, we introduce some of the performance optimization techniques that are commonly used for deep learning on edge devices.

2.1 Diffusion Models

Diffusion models are a recent type of deep generative model that have been successful in image synthesis. They learn to transform random noise into a desired data distribution by iteratively applying an $\mathcal{R}^N \rightarrow \mathcal{R}^N$ denoising neural network (a U-Net). This process is illustrated in Fig 1.

Diffusion models can require upto 1000 iterations of U-Net inference, making them computationally expensive. Most previous work in optimizing diffusion models focus on modifying the architecture or the sampling process. Karras et al. [2] proposed advanced sampling techniques (known as "schedulers") that reduce the number of iterations required to produce high-quality samples. Rombach et al. [3] proposed latent diffusion models (LDMs) which address expensive sampling by performing denoising steps in the compressed latent space of an encoder network. Concretely, the initial image is projected into the latent space once, and the U-Net denoiser is iteratively applied in latent space rather than pixel space. The final latent image is projected back into pixel space by a decoder network. The LDM architecture is illustrated in Figure 2.

2.2 Edge Computing

Edge computing is a distributed computing paradigm that allows data processing to be performed closer to the source of data, typically at or near the edge of the network. This approach has gained traction in recent years due to the proliferation of Internet of Things (IoT) devices, which generate large amounts of data that can overwhelm traditional cloud-based computing architectures [4]. Particularly interesting to us are two potential benefits of edge computing:

Decreased workload on the server. Processing expensive requests such as diffusion model inference on servers can overwhelm it. Therefore, processing them on the edge device can help scale the use of diffusion models and reduce server unavailability.

Decreased energy cost. Edge devices usually consume far less power than large GPUs on the server. This power efficiency can be used to save the server cost, and ultimately the cost to the end user.

We focus on the following two edge devices:

Jetson Nano. Our Nvidia Jetson Nano has a 128-core Maxwell family GPU, a Quad-core ARM A57 CPU and 4GB shared memory. The detailed specifications can be found at `https://developer.Nvidia.com/embedded/jetson-nano-developer-kit`.

Android smartphone. Our device is a Samsung Galaxy S22 with 8GB RAM running Android 13, with a Qualcomm SM8450 Snapdragon 8 Gen 1 chip containing an 8-core Arm-v9 architecture CPU. The detailed specifications can be found at `https://www.gsmarena.com/samsung_galaxy_s22_5g-11253.php`.

Benchmarking and optimizing the performance of deep learning on edge devices is an active area of research because these devices have limited computational resources and battery life. Performance metrics such as inference latency, memory cost, and power consumption are all important.

Baller et al. [5], Süzen et al. [6], and Holly et al. [7] benchmarked the inference time, power, and energy consumption of deep neural networks on various embedded system boards, including the Jetson Nano. Similarly, Ignatov et al. [8] benchmarked the inference latency of deep neural networks on various Android smartphones.

According to a recent blog post, engineers at Qualcomm achieved the first ever locally-run demonstration of Stable Diffusion, a 890M parameter latent diffusion model, on an Android smartphone [9]. However, they do not make code or a demo publicly available, nor do they provide a detailed measure of performance metrics. To the best of our knowledge, the performance of diffusion models, including inference latency, memory, and energy consumption, has not been benchmarked or optimized on the Nvidia Jetson Nano or Android smartphones.

2.3 Performance Optimization

Some common neural network optimization techniques for edge devices include:

1. *Model quantization*: The weights of the neural network are converted to lower precision datatypes (16-bit, 8-bit or even 4-bit), which significantly reduces the memory requirement and compute cost. The quantization can be done post-training, or it can be trained in a quantization-aware manner [10].
2. *Operator fusion*: Common sequences of operators, such as convolution-batchnorm-relu, can be fused into single operators, which can improve data re-use and reduce overhead.
3. *Gradient checkpointing* (for training only): During the forward pass, intermediate values can be removed from memory and be recomputed when they are needed in the backward pass. This trades-off memory for compute.

3 Methodology

We implement and evaluate the performance of a diffusion model on an Nvidia Jetson Nano and an Android Smartphone. We compare this performance with the performance of the same LDM on an Nvidia RTX 3070 laptop GPU. Furthermore, we adopt optimization techniques such as model quantization and operator fusion for inference.

3.1 Model

We adopt the architecture-related optimizations described in Section 2.1: we use a 330M-parameter latent diffusion model (LDM) pre-trained for unconditional image generation on the CelebA HQ dataset, and we use the DDIM scheduler which has been shown to require fewer sampling steps compared to other schedulers. We refactor the implementation available in the open-source Huggingface Diffusers library [11] so that inference on all devices and all optimizations are run using a common model code with no extra dependencies other than PyTorch.

3.2 Jetson Nano

This GitHub repo shows how to run Stable Diffusion on the Nvidia Jetson platform. However this was implemented for the Jetson Orin and Xavier platforms which have much more memory and house the Ampere family of GPUs. For inference and evaluation on the Jetson Nano we perform the following steps:

- First, we flash the Jetpack 4.6.1 package for the Jetson Nano on a 128 GB SanDisk micro-SD card and setup the Ubuntu 18.04 environment on the device.
- We add network routes in order to SSH into the device while running in the headless-mode. Running in the headless mode conserves energy and RAM to render the display which is essential for edge devices.
- Next, we create a custom Docker image [12] using the NGC[13] container for PyTorch 1.10 as the base image. We further install the following libraries to enable inferencing and evaluations: 1) libprotobuf-dev 2) protobuf-compiler 3) protobuf 3.19.6 (python3) 4) onnx 1.13.1 (python3) 5) tensorrt 8.2.2.1 (python3) 6) onnxruntime-gpu 1.11.0 (python3)
- Then we run a device specific evaluation pipeline (Section 3.6) to extract performance metrics for the baseline as well as the different levels of optimizations specified in Section 3.5

3.3 Android Smartphone

- We set up a Linux environment with access to the command line inside the Termux terminal emulator.
- We obtain and install PyTorch binaries for the Arm-v9 (*aarch64*) architecture from the official repository, enabling PyTorch to run on the smartphone CPU.
- We increase the amount of memory available by allocating additional swap memory. While this may slow down the model inference, we found it to be necessary to ensure successful execution.
- We run the model inference using `float32` precision since certain `float16` operations are not implemented on the CPU.
- We measure memory usage using simple command line utilities and measure power consumption by measuring battery discharge rate (in mA).

3.4 Metrics

In order to evaluate the performance, we focus on the following three metrics:

- *Inference Latency*: Computed as the difference between the start and end of the `forward()` call to the model. Reported in seconds (s)
- *Normalized Energy Cost*: Computed as the `average power` $\times$ `latency` captured during the inference. Reported in Joules (J)
- *Memory Cost*: The peak RAM consumption during inference. Reported in megabytes (MB)

3.5 Optimizations

In this project, we focus only on inference using a pretrained model, and do not train the model ourselves. Therefore, we perform optimizations on the pretrained model using various inference engines. In this section, we describe the levels of performance optimizations provided by the inference engines that we utilize for this project.

Baseline. We refer to no level of optimization and `float32` precision as the Baseline.

16-bit. This level of optimization only includes model quantization so that the inference is performed with `float16` precision.

JIT (32-bit). This level of optimization optimizes the performance of the model using the TorchScript Just-In-Time (JIT) compiler which traces the eager-mode model code to lower it to an intermediate representation (IR).

JIT (16-bit). The model is traced and compiled after being quantized to `float16` precision.

ONNX Runtime. The Open Neural Network Exchange (ONNX) Runtime is a runtime designed for machine learning acceleration, including training and inference. It traces the model, identifying opportunities for optimizations. To use this runtime, we first export the PyTorch model into the ONNX format using `torch.onnx.export`. Then, we simplify model graph using the *onnx-simplifier* package which removes redundant nodes by replacing them with their constant outputs (a.k.a constant folding). An example of this operations is shown in Fig. 3. Subsequently, we create an inference session using the ONNX Runtime framework by importing the simplified model and performing semantics-preserving graph-level optimizations as well as layout optimization (grouping contiguous data channels together to enable parallelization on the GPU cores). Finally, we perform inference using this inference session on a random input noise vector.

TensorRT. Similar to ONNX, TensorRT is an inference engine designed for Nvidia GPUs. It performs several optimizations including quantization, layer and tensor fusion, kernel auto-tuning, dynamic tensor memory, and multi-stream execution. This is the highest level of optimization in our project. To use this inference engine, we export the U-Net of the diffusion model to the ONNX format. Subsequently, we run the `trtexec` module provided by the *tensorrt* package which applies several optimizations as mentioned above to create a TRT engine file. The next step is to convert the TRT engine to *TorchScript* file format using the `torchtrtc` tool provided by the *torch-tensorrt* package create by Nvidia. We can then easily import this optimized U-Net using `torch.jit.load` into the PyTorch inference pipeline.

3.6 Evaluation Script

We developed a modular evaluation script to collect the evaluation metrics using the methods specific to a device. For example, we used the script on the Jetson Nano to collect the power using the `tegrastats` command line utility, inference latency using the CUDA-CPU synchronized time between the start and end of the `forward()` call to the model, and peak memory cost using `trtexec`. It consists of three components:

- **readers.py**: Defines a `Reader` class that makes use of command line tools to read hardware sensor values to extract power and memory and is injected into the inference pipeline to extract the latency

	Original Model	Simplified Model
Add	201	185
Cast	172	156
Concat	14	14
Constant	471	0
Conv	67	67
Cos	1	1
Div	38	0
Gemm	24	24
InstanceNormalization	61	61
MatMul	96	64
Mul	126	125
Reshape	282	282
Resize	3	3
Shape	61	0
Sigmoid	47	47
Sin	1	1
Slice	2	2
Softmax	16	16
Split	0	16
Transpose	112	112
Unsqueeze	45	23
Model Size	677.7MiB	523.2MiB

Figure 3: Simplifying the exported ONNX computation graph for the UNet of the diffusion pipeline.

- **evaluation.py**: Defines an `Evaluator` class that runs the inference for different optimization strategy
- A main script called **run_evaluation.py** that detects the underlying hardware and compiles all the different optimization strategies with hardware specific `Reader`'s

The script is publicly available here.

4 Results

4.1 Comparison of Optimizations

The following three tables show the performance on each hardware for each of the optimizations applied, along with their respective baselines.

Table 1: Nvidia RTX 3070

Model	Latency (s)	Peak Memory (MiB)	Power (W)	Energy Consumed (J)	Speedup
Baseline	1.17	2032.2	101.43	118.6731	1.00
16-Bit	0.903	1201.9	78.95	71.29185	1.30
JIT (16-Bit)	0.795	1248.5	75.21	59.79195	1.47
JIT (32-Bit)	1.315	1793.1	93.944	123.53636	0.89
ONNX Runtime	0.88	3921.8	114.53	100.7864	1.33
TensorRT	0.25	4745.2	66.81	16.7025	4.68

Evaluating optimized U-Net on RTX 3070

The procedure adopted to optimize is summarized in Algorithm 1. A key point in this strategy was the usage of the `Torch-TensorRT` library. `Torch-TensorRT` is an Ahead-of-Time compiler targeting NVIDIA GPUs via NVIDIA's TensorRT Deep Learning Optimizer and Runtime [14]. An

Table 2: Nvidia Jetson Nano

Model	Latency (s)	Peak Memory (MiB)	Power (W)	Energy Consumed (J)	Speedup
Baseline	93.22	2473.2	3.73	347.7106	1.00
16-Bit	99.03	1253.6	3.75	371.3625	0.94
JIT (16-Bit)	92.96	2869.4	3.60	334.66	1.0028
JIT (32-Bit)	88.79	2568.9	3.66	324.97	1.05

Table 3: Android smartphone

Model	Latency (s)	Peak Memory (MiB)	Power (W)	Energy Consumed (J)
Baseline	160.5	2379	3.52	564.96

AOT (Ahead-of-Time) compiler is a type of compiler that compiles source code into machine code before the code is executed, as opposed to a JIT (Just-in-Time) compiler that compiles code at runtime. In other words, AOT compilers generate executable files that can be run directly on a target system, without requiring the source code or a virtual machine to be present. This was one of the big reasons for the speedup in the inference.

Optimizing the UNet using the TensorRT framework gave promising results. There was an overall 4.7X speedup in inference time. Figure 4a provides a breakdown of the improvement in latency for the denoising loop as well as the decoding phase of the pipeline before and after the optimization. The denoising loop had a 3.4X speedup while the decoding loop had a 1.7X speedup. There was also a 36% reduction in power consumption resulting in 86% reduction in total energy consumed (Fig 4b

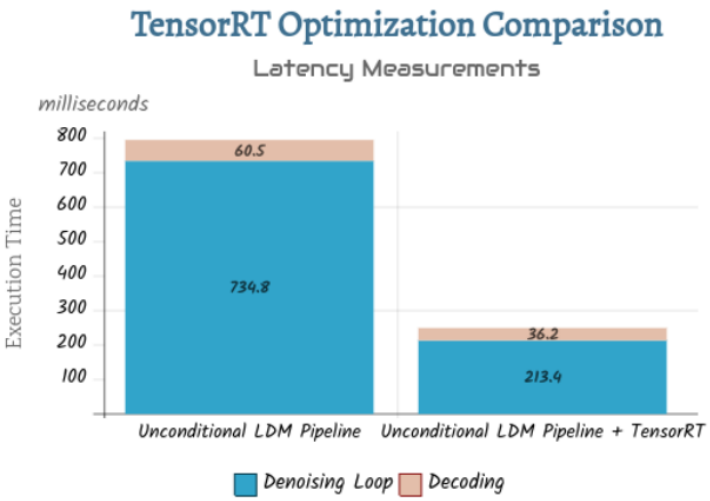
However, there was a 37% increase in the memory usage after the optimization as observed in Fig 4c

Algorithm 1 UNet Optimization (TensorRT)

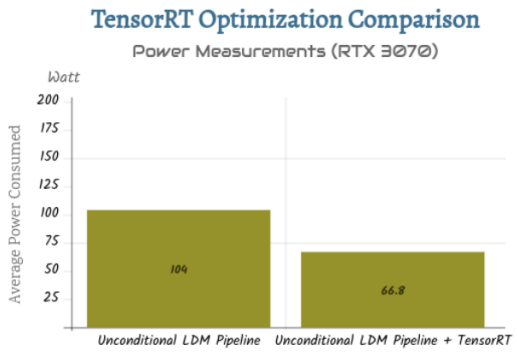
Initialize the pipeline:
diffusion_pipeline ← LDMPipeline()
DTYPE ← torch.float16 ▷ Define fp16 quantization
DEVICE ← torch.device('cuda') ▷ Define CUDA runtime
diffusion_pipeline ← diffusion_pipeline.to(device=DEVICE, dtype=DTYPE) ▷ Quantize and move pipeline to GPU
diffusion_pipeline.export_unet_to_onnx() ▷ Export the UNet to ONNX format
Run the following shell commands:
`onnxsim uldm_unet_fp16.onnx uldm_unet_fp16_sim.onnx` ▷ Removes redundant nodes using ConstantFolding
`trtexec --onnx=uldm_unet_fp16_sim.onnx --saveEngine=uldm_unet_fp16_sim.trt --buildOnly` ▷ Performs Layer and Tensor fusion using cuBLAS, cuBLASLT and cuDNN tactics
`torchtrtc uldm_unet_fp16_sim.trt uldm_unet_fp16_sim.ts --embed-engine --device-type=gpu` ▷ AOT compile the model for PyTorch
Load the optimized UNet in the pipeline:
diffusion_pipeline.unet ← torch.jit.load(uldm_unet_fp16_sim.ts) ▷ Load the TorchScript UNet model and move to GPU
diffusion_pipeline.unet.to(dtype=DTYPE, device=DEVICE) ▷ Move the UNet to the GPU

4.2 Summary of Results

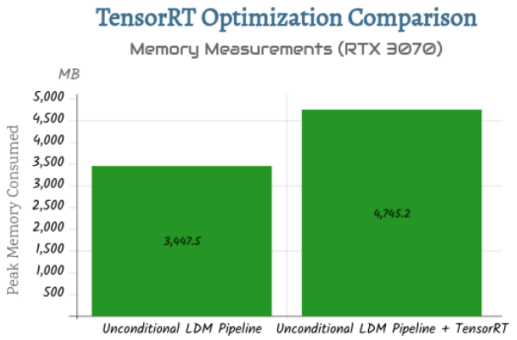
The table 4 summarizes the performance on the three hardware devices, based on several metrics. In terms of latency, the RTX 3070 outperforms both Jetson Nano and Android, with a latency of 1.17 seconds, while Jetson Nano and Android have latencies of 93.22 and 160.51 seconds, respectively. However, the RTX 3070 consumes significantly more power, with a power consumption of 101.43 watts, compared to Jetson Nano and Android, which consume only 3.73 and 3.52 watts, respectively.



(a) Latency comparison



(b) Power comparison



(c) Memory comparison

Figure 4: Performance comparison for UNet optimization using TensorRT on RTX 3070

As a result, the RTX 3070 has a total energy consumption of 118.67 joules, while Jetson Nano and Android have total energy consumptions of 347.71 and 564.96 joules, respectively.

We also calculate the relative latency and relative energy metrics, where the RTX 3070 serves as the reference point. It is observed that the Jetson Nano is 79.68 times slower than the RTX 3070, while Android is 137.18 times slower. Similarly, Jetson Nano consumes 2.93 times more energy, while Android consumes 4.76 times more energy than the RTX 3070.

Overall, it is worth noting that even though the RTX 3070 consumes more instantaneous power, due to the quicker inference, the total energy consumption per sample is lower compared to the Jetson Nano and Android. Therefore, the choice of hardware device depends on the specific use case, taking into consideration the trade-off between performance, power and energy consumption.

As a qualitative comparison of the performance of the diffusion model on each device, Fig 5 shows an image generated by the diffusion model on each device. We expect that there should be no difference in the quality of the images between devices with the same level of optimization, and that is what we observe.

Table 4: Summary of performance results

Metric	**RTX 3070**	**Jetson Nano**	**Android smartphone**
Latency (s)	1.17	93.22	160.51
Peak Memory (MB)	2032	2473	2379
Power (W)	101.43	3.73	3.52
Total Energy (J)	118.67	347.71	564.96
Relative Latency	1X	79.68X	137.18X
Relative Energy	1X	2.93X	4.76X

(a) A face image generated by the diffusion model running on the Android smartphone.

(b) A face image generated by the diffusion model running on the laptop GPU.

(c) A face image generated by the diffusion model running on the Jetson Nano.

Figure 5: The quality of the generated images remain the same when running on the edge devices.

5 Challenges and Limitations

Memory Constraints:

During initial phases of the project we faced severe memory constraints while running the larger diffusion models such as Stable Diffusion. The Jetson Nano comes with only 4GB shared-RAM and we quickly realized that running Stable Diffusion was not an option as we weren't able to load the pretrained weights. We therefore decided to switch to using the unconditional LDM model pretrained on the CelebA-256 dataset and were able to successfully perform inference on the Jetson.

However, while adopting some of the optimization strategies, specifically while creating the TensorRT engine for the entire diffusion pipeline, the process was getting `OOM Killed` before it was able to create the engine. We tried optimizing only the UNet (denoising loop) of the pipeline and were successfully able to do so on the RTX. However, we were still getting out-of-memory errors on the Jetson for just the U-Net as well. We were therefore unable to perform this optimization strategy on the Jetson Nano even though it gave us the best performance on the RTX.

Similar issues were faced on the Android device. While the baseline model ran in plain PyTorch, running a traced version of the model within a TorchScript or ONNX runtime introduced additional memory overheads which caused it to fail.

Unavailability of hardware-specific libraries:

The Jetson Nano runs on Jetpack 4.6.1 and supports Python 3.6.9. Many libraries such as *TensorRt* and *Torch-Tensorrt* do not provide pre-compiled packages for this version of Python. Since our evaluation methodology relies heavily on using these libraries, we had to compile the libraries from the source code on the device and mount the package files onto the docker image. This required a lot of additional time and effort in terms of ensuring that all the system binary-dependencies are satisfied during compilation and that the package runs as expected inside the container environment.

6 Future Work

On March 21st 2023, Nvidia launched the Jetson Orin Nano developer kit. The Jetson Orin Nano comes with the Nvidia Ampere architecture and has 1024 CUDA core and 32 Tensor cores. It is also equipped with a 6-core ARM Cortex CPU and 8GB RAM. This architecture is capable of supporting TensorRT optimization as well as 8-bit quantization and is therefore the most suitable direction for the future work on running optimized diffusion models on edge devices. It also has the potential

of running the large 890M parameter Stable Diffusion model, which is an important goal for future work.

The most promising line of work for optimizing diffusion models for smartphones is performing 8-bit quantization. It has been observed that the distribution of activations changes over the denoising iterations. Therefore, standard post-training quantization techniques that assume a static distribution do not work with diffusion models [15]. Currently there are no successful 8-bit quantization attempts that achieve performance gains, except for the Qualcomm demonstration which has not been made publicly available to be verified. Future work in this direction will make high-fidelity diffusion models possible to run locally on most smartphones.

7 Conclusion

In this project, we benchmark the inference performance of a latent diffusion model with 330M parameters on the Nvidia Jetson Nano and an Android smartphone, and compare it with the performance of the model on the Nvidia RTX 3070 laptop GPU. Our results show that the inference latency on the Jetson Nano and the Android is 79.68X and 137.18X the inference latency on the RTX 3070. Moreover, despite consuming significantly lower instantaneous power, the Jetson Nano and the Android consume 2.93X and 4.76X more energy per sample, due to slower inference. We also show the use of inference engines such as ONNX Runtime and TensorRT to perform optimizations on the RTX 3070. We empirically evaluate the performance improvements from these optimizations on the RTX 3070, and our results show that TensorRT performs the best optimization, with a 4.68X speedup and 86% reduction in total energy consumption on the RTX 3070, albeit with a higher peak memory cost. We have published our Evaluation Script and Docker Images for the Jetson Nano for use by the community. The newly released Jetson Orin Nano addresses many of the challenges we encountered, and we expect that it can be used to significantly extend this project and its impact.

References

[1] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020.

[2] T. Karras, M. Aittala, T. Aila, and S. Laine. Elucidating the design space of diffusion-based generative models. *arXiv preprint arXiv:2206.00364*, 2022.

[3] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

[4] W. Shi, J. Cao, and Q. Zhang. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[5] S. P. Baller, A. Jindal, M. Chadha, and M. Gerndt. Deepedgebench: Benchmarking deep neural networks on edge devices. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 20–30, 2021. doi:10.1109/IC2E52221.2021.00016.

[6] A. A. Süzen, B. Duman, and B. Şen. Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn. In *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, pages 1–5, 2020. doi:10.1109/HORA49412.2020.9152915.

[7] S. Holly, A. Wendt, and M. Lechner. Profiling energy consumption of deep neural networks on nvidia jetson nano. In *2020 11th International Green and Sustainable Computing Workshops (IGSC)*, pages 1–6, 2020. doi:10.1109/IGSC51522.2020.9290876.

[8] A. D. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. V. Gool. Ai benchmark: Running deep neural networks on android smartphones. In *ECCV Workshops*, 2018.

[9] Qualcomm. World's first on-device demonstration of stable diffusion on android, February 2023. URL https://www.qualcomm.com/news/onq/2023/02/worlds-first-on-device-demonstration-of-stable-diffusion-on-android.

[10] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[11] P. von Platen, S. Patil, A. Lozhkov, P. Cuenca, N. Lambert, K. Rasul, M. Davaadorj, and T. Wolf. Diffusers: State-of-the-art diffusion models. https://github.com/huggingface/diffusers, 2022.

[12] S. Bagri. csc2231 - docker hub. https://hub.docker.com/repository/docker/trapdoor20/csc2231/general, year accessed.

[13] NVIDIA. l4t-pytorch - ngc catalog. https://catalog.ngc.nvidia.com/orgs/nvidia/containers/l4t-pytorch, year accessed.

[14] PyTorch. Tensorrt integration for pytorch. https://github.com/pytorch/TensorRT, 2021.

[15] Y. Shang, Z. Yuan, B. Xie, B. Wu, and Y. Yan. Post-training quantization on diffusion models. In *CVPR*, 2023.

Appendix A: Group Contributions

Akash Haridas: Implemented the diffusion model pipeline and refactored the code such that it runs on the 3 different devices and enables JIT compilation and optimization. Responsible for running and evaluating the model on the Android device. Contributed towards the poster and report.

Jazib Ahmad: Contributed towards the development and implementation of the evaluation method on the Nvidia Jetson Nano, the analysis of the results, and the preparation of the project report and poster.

Sumant Bagri: Contributed towards the environment setup on the Jetson Nano, containerization of the diffusion model runtime in docker as well as creating a modular evaluation pipeline that can be extended to newer Jetson devices as well as other edge devices. Responsible for running the evaluation scheme, extracting metrics from RTX 3070 and Jetson Nano and collating the final results and methodology for the report and poster.