Seminar - Thrustworthy AI Models and Systems : Project Report Extending MixDQ model performances

Fabia Schreyer [1 *] **Loan Strübi** [1 *]

Abstract

Few-step T2I diffusion models like SDXL-Turbo are fast but consume a lot of memory and are not addapted to low-end GPUs. Post-training quatization, like in the MixDQ scheme, is a promising way to optain a good image quality and text-image alignment while keeping the memory use and latency low. Our objectives are to reproduce the core results and observations of the MixDQ paper by comparing FP16 and MixDQ on different metrics. We'll them extend the analysis to a system level, by experimenting on quantization and resolution control to see the effects on latency, VRAM usage and quality in a queue-based service. In order to do that, we generated 2048 (FP16, MixDQ 50/50) based on the COCO images Dataset using SDXL-Turbo and the same prompts as The paper. These images are then evaluated with FID, CLIP and ImageReward. The system part looks at multiple bit-widths and resolution. It then simulate a queue-server with : static resolution vs. adaptative resolution controller that switches between resolution based on queue length.

Link to our github repo

1. Research problem

1.1. MixDQ paper

Few-step text-to-images diffusion models like SDXL-Turbo are attractive for image generation. They have fewer denoising steps than other models,like Stable diffusion v1.x/v2.x or the non-Turbo version of SDXL, which makes them faster (1-2 step for SDXL-Turbo vs. 20-50 for the others).
One of their main issue is the memory usage which, despite the fewer denoising steps, is still high and limits their deployment on low-end GPUs. A solution to that is post-training quantization (PTQ) to reduce the memory by representing weights and/or activation in INT4/INT8 instead of FP16.
Most works prior to the MixDQ paper concerned multi-step diffusion or other models and not few-step T2I diffusion. The ones targeting few-step models generally failed to preserve both visual quality and text-image alignment under low-bit quantization.
The core objective of the MixDQ paper is therefore to design a mixed-percision PTQ scheme for few-step T2I diffusion perserving image quality and text-image alignment while reducing memory and latency.
Since few-step models don't have extra steps to take care of quantization noise, they are more sensitive to quantization than multi-step models. Some Layer layer are very sensitive and risk becoming quantization bottlenecks. The way you quantize different parts of the model affects image content and visual quality differently. The BOS token is also a big source of sensitivity.
To answers these issues in the MixDQ paper they implemented BOS-aware text embedding quantization which treats the BOS token separately. They also analized which layers are more important for which type of metric and used a sensitivity analysis to give a higher precision where it matters more.
In their contribution section they emphasized the difficulty of quantizing few-step diffusion while preseving text-image alignment. (Zaho, 2024)

1.2. What we plan to do

In the Optimization Outcomes part of the project we'll experiment on the lactency/image-quality equilibrium. For that we'll simulate service in a queue-based system with different load conditions. This will allow us to see how the quantization and resolution affect time, memory usage, image-quality and in the end find the best suited configuration for a certain configuration.

Seminary - Thrustworthy AI Models and Systems [1]Department of Informatics, University of Neuchâtel, Neuchâtel, Switzerland. Correspondence to: <
>.

2. Reproducting results

In order to reproduct the results from the MixDQ paper, we must first generate images, and then make an evaluation script

2.1. Generating images

In order to reproduct the results from the paper, we must generate 2048 images :
1024 with FP16,
1024 with MixDQ (W8A8)

Using python scripts to generate images based on the text prompts. After that we'll be able to evaluate them, for that we need to store the images accordingly. Following are the bash commands requiered to do so.

2.1.1. FP16 GENERATION

Original SDXL-Turbo math, everything in 16 bits float

Bash 1 Simplified command for FP16 image generation

```
CUDA_VISIBLE_DEVICES=0 python
scripts/txt2img.py \
  --config .../sdxl_turbo.yaml \
  --base_path logs/sdxl_fp_eval_big \
  --num_imgs 1024 \
  --batch_size 4 \
  --fp16
```

- `CUDA_VISIBLE_DEVICES` : GPU number 0 used for this run
- `python` : to run txt2img.py
- `txt2img.py` : the file that generates the images based on the text prompts
- `--config` : base sdxl_turbo,
- `--base_path` : saves the outputs in `sdxl_fp_eval_big`
- `--num_imgs`: number of images genrated, here 1024, same amounth ass in the paper
- `--batch_size` : 4 is chosen because higher uses to much memory for the VM, and often caused the command to fail.
- `--fp16` : Use 16-bit floating point

2.1.2. MIXDQ GENERATION

Bash 2 Simplified command for MixDQ image generation

```
CUDA_VISIBLE_DEVICES=0 python
scripts/quant_txt2img.py \
  --base_path ".../sdxl_mixdq_eval" \
  --image_folder ".../sdxl_mixdq_eval_images"\
  --batch_size 4 \
  --num_imgs 1024 \
  --config_weight_mp "$WEIGHT_MP_CFG" \
  --config_act_mp "$ACT_MP_CFG" \
  --act_protect "$ACT_PROTECT" \
  --fp16
```

- `CUDA_VISIBLE_DEVICE = 0 python` : same as for FP16 genereation, this tells which device is used (here GPU # 0) and calls python to run the `--quand_txt2img.py` file.
- `--quant_txt2img.py` : runs the python files, loading the MixDQ-quantized SDXL model. Generates images based on text prompts.
- `base_path` & `--image_folder` : the first one is for the evaluation, the second one stores the images
- `--batch_size, --num_imgs` : same values as in the FP16 generation, to match the paper's generation while keeping the memory use at a stable amount.
- `--config.weight_mp` : path to the config file that describes the weights in mixed-precision and quatization settings for MixDQ, such as the bit-width per layer.
- `--config.act_mp` : path to the config file that describes the how activations are quantized per layer.
- `--act_protect` : config file that specifies which layers/activations should be protected, if they should be kept at a higher precision or be heavily quantized.
- `--fp16` : enable FP16 computation for the underlying model, over which MixDQ then applies its quantization scheme.

Both Command generate 1024 images with FP16 and MixDQ and are based on the same prompts as in the MixDQ paper. The files in which the images are stored will be used later during the evaluation part.

2.1.3. EXAMPLES OF GENERATED IMAGES

Results for the prompt : ”A cat on a toilet seat of some sort”

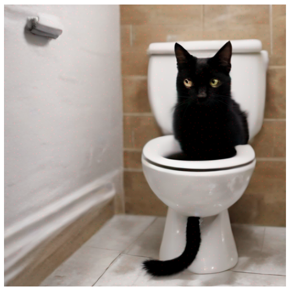


Figure 1. FP16 image outcome

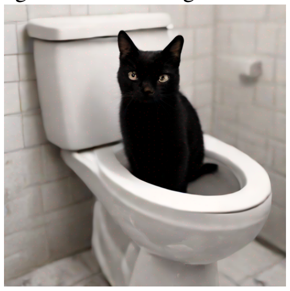


Figure 2. MixDQ image outcome

Despite the sillyness of this prompt, these images show that MixDQ's results might be better. In the FP16 outcome, the cat's tail goes through the toilet bowl, which does not make sense. Also, the cat is less noisy in MixDQ's image.

2.2. Evaluation Metrics

All three of our evaluation metrics are in the file `eval_fp16_mixdq`, are called via a bash command to be executed.

2.2.1. FID

We used the `cleanfid` library for our FID score, It's implementation is the following :

Python 3 FID in our eval script

```
Input: ref_folder, fp16_folder, mixdq_folder, dev_str
Output: fid_fp16, fid_mixdq
Function FID(ref_dir, gen_dir, device)
    score ← CleanFIDCompute(
        ref_dir, gen_dir, device, num_workers
        )
    return score
End Function
fid_fp16 ← FID(ref_folder, fp16_folder, dev_str)
fid_mixdq ← FID(ref_folder, mixdq_folder, dev_str)
```

The `FID` function take three arguments, `ref_dir`, `gen_dir` and `device` which are, respectively :

`ref_dir` : the folder containing the reference images
`gen_dir` : the folder containing the generated images
`device` : e.g. `cuda` or `cpu`

It then calls `CleanFIDComupte` to get the FID score. For that it also uses a `num_worker` which is the number of dataloader worker.
The outputs are the FID scores of our PF16 and MixDQ images.

This function is called twice, once for the FP16 folder and a second time for the MixDQ folder.

2.2.2. CLIP

We used openai's `clip` metric like they seem to have done in the paper

Python 4 CLIP in our eval script

```
Input: fp16_imgs, mixdq_imgs, prompts, clip_model, clip_preprocess, device, batch_size
Output: clip_mean_fp16, clip_std_fp16, clip_mean_mixdq, clip_std_mixdq
Function CLIP_SCORES(image_paths, prompts)
    sims ← CLIPComputeSimilarities(
        image_paths, prompts, clip_model,
        clip_preprocess, device, batch_size
        )
    mean ← Mean(sims)
    std ← Std(sims)
    return mean, std
End Function
clip_mean_fp16, clip_std_fp16 ← CLIP_SCORES(fp16_imgs, prompts)
clip_mean_mixdq, clip_std_mixdq ← CLIP_SCORES(mixdq_imgs, prompts)
```

Our function takes the FP16 and MixDQ images, text prompts, clip model and preprocess function, device and batch size as inputs.
The clip model and preprocess function are the closest ones we found to match the ones used in the MixDQ paper (CLIP ViT-L/14).

The `CLIPCOMPUTESIMILARITIES` does the image preprocessing by loading each image and converting it to RGB, after that it applies the `clip_preprocess` and stack the processed images into a tensor which is then

moved to the device.
It then tokenize the corresponding prompts and move them to the device. The images and texts prompts are then embedded and the embedding are normalized.
For the CLIP score, it computes the dot product between the normalized image and text embeddings and collects the scores.

The output is the mean CLIP score and it's standard deviation.

2.2.3. IMAGEREWARD

Python 5 IR in our eval script

```
Input: fp16_imgs, mixdq_imgs, prompts,
          irmodel, batch_size
Output: ir_mean_fp16, ir_std_fp16,
          ir_mean_mixdq, ir_std_mixdq
Function IMAGEREWARD_SCORES(image_paths, prompts)
    scores ← IRComputeScores(
      image_paths, prompts, irmodel, batch_size
      )
    mean ← Mean(scores)
    std ← Std(scores)
    return mean, std
End Function
ir_mean_fp16, ir_std_fp16 ←
IMAGEREWARD_SCORES(fp16_imgs, prompts)
ir_mean_mixdq, ir_std_mixdq ←
IMAGEREWARD_SCORES(mixdq_imgs, prompts)
```

Our function's input are the following : The FP16 and MixDQ images, the text prompts, the `irmodel` and batch size.
As for the CLIP function we used the closest one we could find to the one used in the paper which is `ImageReward-v1.0`.

`IRCOMPUTESCORES` first checks if every image as a matching prompt. Then it loops over the dataset in chunks base of `batch_size` and for every pair of image-prompt it calls the IR score and stores it in a list. the mean and std of the scores are then computed.

The output is the mean IR score and it standard deviation.

2.2.4. EVALUATION COMMAND

The evaluation python script is called by the following command :

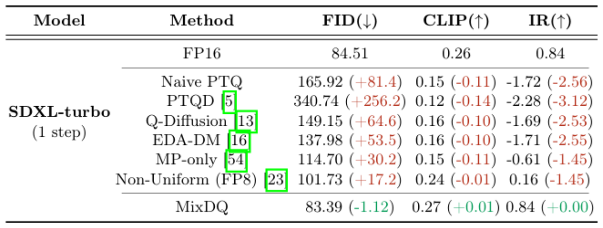
Bash 6 Command for FP16 vs. MixDQ evaluation

```
python
scripts/eval_fp16_mixdq_cleanfid.py \
  --ref_folder .../val2014 \
  --fp16_folder .../"FP16_gen_imgs" \
  --mixdq_folder .../"mixdq_gen_imgs"\
  --prompts_file .../prompts.txt \
  --batch_size 16 \
  --device cuda \
  --num_samples 1024
```

- `python and eval_fp16_mixdq_cleanfid.py` : run the evaluation script using python.
- `--ref_folder` : path to the images used as the real distribution for the FID computation.
- `--fp16_folder and --mixdq_folder` : paths to the folders containing the FP16 and MixDQ generated images.
- `--prompts_file` : path to the file containing the text prompts.
- `-- batch_size, --device and --num_samples` : makes sure we use cuda, and have the right amount of samples.

2.3. Evaluations Results

Our objective was to match the results of the following table for FP16 and MixDQ[1]:



Model	Method	FID(↓)	CLIP(↑)	IR(↑)
SDXL-turbo (1 step)	FP16	84.51	0.26	0.84
	Naive PTQ	165.92 (+81.4)	0.15 (-0.11)	-1.72 (-2.56)
	PTQD [5]	340.74 (+256.2)	0.12 (-0.14)	-2.28 (-3.12)
	Q-Diffusion [13]	149.15 (+64.6)	0.16 (-0.10)	-1.69 (-2.53)
	EDA-DM [16]	137.98 (+53.5)	0.16 (-0.10)	-1.71 (-2.55)
	MP-only [54]	114.70 (+30.2)	0.15 (-0.11)	-0.61 (-1.45)
	Non-Uniform (FP8) [23]	101.73 (+17.2)	0.24 (-0.01)	0.16 (-1.45)
	MixDQ	83.39 (-1.12)	0.27 (+0.01)	0.84 (+0.00)

Figure 3. MixDQ paper's results table

The MixDQ paper doesn't specify exactly which versions of FID, CLIP and ImageReward they used so our score may differ from the paper results. But the main objective of these metrics scores is the see the difference between FP16 and MixDQ results.
In addition to that we might have a problem with the way text prompts and images are paired, our results values were mediocre but we observe the same

[1]Cornell University, Tianchen Zhao, Thu, 30 May 2024 01:51:10 UTC https://arxiv.org/abs/2405.17873

behaviors as in the paper, FID is slightly better, CLIP score too and ImageReward remain almost exactly the same.

Our results are the following :

Results for our experiments			
eval metric	FID	CLIP	IR
FP16	87.44	0.072	-2.221
MixDQ	87.01 (-0.43)	0.074 (+0.002)	-2.2211

Both CLIP and IR are extremly low, to be sure that the problem came from the generated images - text promts pairs, we ran our evaluation on the COCO images and their prompts and the results were similar to the ones in the MixDQ paper, since we didn't know how much of `txt2img.py` or `quant_txt2img.py` we could modify and because of a lack of time we went with these results.

3. Optimization outcomes

In the following part of the project we get through simple FP16 and MixDQ quality metrics with SDXL-Turbo as a service in a queue-based system. The objectif was to understand how quantization and resolution control can affect latency, VRAM usage and output quality, for then identify the best configuration under different load conditions.

3.1. Experimental setup

We choose to work with SDXL-Turbo (1 step), rather than the full SDXL model (20-30 step), because it's still an SDXL architecture but distilled into a few steps or one step latent consistency model. The utilisation of SDXL-Turbo has two advantages for the project:

1. The cost per request is low enough that the effects of quantization and resolution control are clearly visible in latency and VRAM.
2. It can make large sweeps over bit-widths, resolutions and load on a single GPU.

Therefore, full SDXL-turbo is treated as a quality reference, and we use Turbo model as our main target for system-level optimisation. To compare a full-precision baseline we choose several quantized variants:

- FP16, SDXL-Tubo with weights and activations in 16bits float point, equivalent to W16A16
- Several bits for MixDQ, Quantized weights of 8, 6 and 4bits for FP16 activations

We evaluate each bit-width on three resolutions - 512, 768 and 1024. With "eval_grid.py", the scripts sweeps over all combinations of bits/resolution.

Table 1. Grid evaluation over 32 prompts: p95 latency, peak VRAM and CLIPScore for SDXL-Turbo at different resolutions and weight bit-widths.

	512^2			768^2			1024^2		
wbits	p95 [ms]	VRAM [GB]	CLIP	p95 [ms]	VRAM [GB]	CLIP	p95 [ms]	VRAM [GB]	CLIP
16	113.5	12.54	0.349	201.3	13.99	0.343	349.2	15.63	0.335
8	1046.9	14.54	0.350	1242.5	15.72	0.343	1620.9	17.36	0.330
6	1000.3	14.55	0.345	1268.3	15.72	0.339	1652.0	17.36	0.321
4	997.1	14.55	0.348	1238.4	15.72	0.341	1650.9	17.36	0.328

The table shows some unexpected results, quantizing SDXL-Turbo with MixDQ does not bring benefit. We can see that FP16 always has the best latency or memory benefits in our implementation. While CLIPScore remains the same across all bit-widths and resolutions, the p95 latency makes a big increase moving from FP16(16bits) to W8/W6/W4. The VRAM also grows by about 1.5 to 2GB at each resolution. This implies that the quantized pipeline (mixed-precision configuration, QuantModel wrapper) dominates any potential gains from lower-precision wights on our setup.

3.2. Resolution control in a queue-based setting

To simulate a real service, we introduce a simple queue-server model where text-to-image requests arrive over time and are processed on a single GPU. The idea is to dynamically adapt the resolution to the load. A controller maps the current queue length to a resolution. How it works ? With thresholds (q_low, q_high). When the queue is quite empty we use 1024 to maximize quality. For medium load we're down to 768, and with high load - 512 to keep a reasonable latency. The configuration is chosen to satisfy a target service-level objective (SLO) on the p95 latency.

With the script load_test_stub.py generates synthetic traffic and compares two policies:

1. **Static resolution**: always generate 1024 images
2. **Adaptative resolution**: using the ResolutionController, with the estimations queue length from the number of requests.

We consider, all requests submitted as fast as possible and Poisson arrivals with exponential inter-arrival times at different rates. For each scenario we record p50 and p95 latency and compare FP16 and W8/6/4A16

3.3. Configuration

1. p95 latency below the chosen SLO in the high-load scenarios,
2. peak VRAM low enough to coexist with other jobs on the same GPU,
3. minimal degradation in CLIPScore and visual quality compared to the FP16 baseline.

3.4. Queuing results

Table 2. Burst workload: p95 response time for FP16 and W8 with static vs. adaptive resolution control.

Concurrency	Mode	Policy	p95 [ms]
1	FP16	static	7477.2
1	FP16	adaptive	7242.7
1	W8	static	9048.7
1	W8	adaptive	9175.5
2	FP16	static	8414.7
2	FP16	adaptive	7740.2
2	W8	static	10891.8
2	W8	adaptive	9663.3

Here, we report p95 response times for bursty workloads at concurrency level c1,2. The adaptive control is almost always better than the static one, showing that our model works well and is effective. FP16 on adaptive reduces p95 latency, especially at c=2, where p95 drops from about 8.414s to 7.74s. However, the W8 configuration is substantially slower in all cases, even with adaptive resolution. Showing that, under burst traffic, a dynamic controller helps, but the quantized model is not as good as FP16 baseline.

Table 3. Poisson arrivals: p95 response time vs. arrival rate λ for FP16 and W8, static vs. adaptive resolution.

λ [req/s]	Mode	Policy	p95 [ms]
1.0	FP16	static	9050.9
1.0	FP16	adaptive	8797.9
1.0	W8	static	11241.7
1.0	W8	adaptive	10480.4
4.0	FP16	static	7789.1
4.0	FP16	adaptive	7412.1
4.0	W8	static	9806.4
4.0	W8	adaptive	9245.6

The table summarizes p95 latency under Poisson arrivals for medium and high load ($\lambda = 1.0$ and 4.0 req/s). Once again, the adaptive policy consistently improves p95 compared to the static 1024 strategy. We can see a difference of between 5% to 10% depending on the load. As the burst scenario, W(8) has systematically higher p95 than FP16. Overall, our queuing experiments support the use of adaptive resolution, but it is not enough to replace the FP16 model with the W8 MixDQ implementation.

4. conclusion

In this project we explore with MixDQ two complementary angles: a reproduction of its evaluations results on SDXL-Turbo, and the implementation of a model that is a queue-based text-to-image service.

On the reproduction side, we generated 1024 FP16 and 1024 MixDQ (W8A8) images from COCO images prompts using SDXL-Turbo. We used FID, CLIP and ImageReward score-metrics to evaluate these images. Although our scores differ from the ones in the MixDQ paper (mainly due to implementation details and potential metric variants) the relative behavior of FP16 vs. MixDQ is consistent. W8A8 improved both FID and CLIP, while IR stay basicaly the same (only +0.0001). This confirms the text-image alignement and image quality adventages of the MixDQ scheme under low-bit quantization on few-step diffusion models.

On the system side, our results were more interesting. The quatized SDXL-Turbo varaints did not perform so well across the many resolutions (512, 768, 1024) and weight bitwidths (W16, W8, W6, W4), it did not deliver the expected latency of VRAM benefits. In our model, FP16 always achived lower p95 latency and had a lower memoty use than the MixDQ models. This could mean that the overhead of the quantization pipeline dominates any savings from the low-precision weights. On the other hand, the adaptive resolution control significantly improved the system performances. Switching between the resolution based on queue length reduced the p95 latency by around 5 to 10% compared to the static 1024-only policy. This under both vurst and poisson workloads, with no radical changes to the model architecture.

Overall, our experiments show that, with our setup, resolution control is currently more efficient than weight quantization for latency and VRAM constraints in SDXL-Turbo-based services. At the same time, quality metrics show that MixDQ remains promising for image-quality.

References

Zaho, T. Mixdq: Memory-efficient few-step text-to-image diffusion models with metric-decoupled mixed precision quantization. Technical report, Tsinghua University, Infinigwnce AI, Haidian District, Beijing, 2024.

A. You *can* have an appendix here.

B. index

Image generation & evaluation - COCO images

```
# FP16
CUDA_VISIBLE_DEVICES=0 python scripts/txt2img.py \
  --config ./configs/stable-diffusion/sdxl_turbo.yaml \
  --base_path logs/sdxl_fp_eval_big \
  --num_imgs 1024 \
  --batch_size 4 \
  --fp16


# MixDQ
WEIGHT_MP_CFG="./mixed_precision_scripts/mixed_percision_config/
sdxl_turbo/final_config/weight/weight_8.00.yaml"
ACT_MP_CFG="./mixed_precision_scripts/mixed_percision_config/
sdxl_turbo/final_config/act/act_7.77.yaml"
ACT_PROTECT="./mixed_precision_scripts/mixed_percision_config/
sdxl_turbo/final_config/act/act_sensitivie_a8_1%.pt"

CUDA_VISIBLE_DEVICES=0 python scripts/quant_txt2img.py \
  --base_path "./logs/sdxl_mixdq_eval" \
  --image_folder "./logs/sdxl_mixdq_eval_images" \
  --batch_size 4 \
  --num_imgs 1024 \
  --config_weight_mp "$WEIGHT_MP_CFG" \
  --config_act_mp "$ACT_MP_CFG" \
  --act_protect "$ACT_PROTECT" \
  --fp16


# Evaluation

python scripts/eval_fp16_mixdq_cleanfid.py \
  --ref_folder ./scripts/utils/val2014 \
  --fp16_folder ./logs/sdxl_fp_eval_big/generated_images \
  --mixdq_folder ./logs/sdxl_mixdq_eval_images \
  --prompts_file ./scripts/utils/prompts.txt \
  --batch_size 16 \
  --device cuda \
  --num_samples 1024
```

Global grid bits/resolutions - grid_bits_res.csv

```
    export PYTHONPATH=".":"$(pwd)/quant_utils"

python scripts/eval_grid.py \
  --config ./configs/stable-diffusion/sdxl_turbo.yaml \
  --ckpt ./logs/sdxl_mixdq_eval/ckpt.pth \
  --prompts_file ./scripts/utils/prompts_32.txt \
  --out_csv ./logs/grid_bits_res_32.csv \
  --res 512 768 1024 \
  --wbits 8 6 4


# FP16
python scripts/latency_probe.py \
  --mode fp16 \
  --base_path ./logs/sdxl_mixdq_eval \
  --res 512 768 1024 \
  --repeats 30 \
  --out ./logs/latency_fp16.csv


# W8A8
# W8A8
python scripts/latency_probe.py \
  --mode w8a8 \
  --base_path ./logs/sdxl_mixdq_eval \
  --config_weight_mp ./mixed_precision_scripts/mixed_percision_config/sdxl_turbo/final_config
  --config_act_mp ./mixed_precision_scripts/mixed_percision_config/sdxl_turbo/final_config/ac
  --act_protect ./mixed_precision_scripts/mixed_percision_config/sdxl_turbo/final_config/act/
  --res 512 768 1024 \
  --repeats 30 \
  --out ./logs/latency_w8a8.csv



# W6A8
python scripts/latency_probe.py \
  --mode w6a8 \
  --base_path ./logs/sdxl_mixdq_eval \
  --config_weight_mp ./mixed_precision_scripts/mixed_percision_config/sdxl_turbo/final_config
  --config_act_mp ./mixed_precision_scripts/mixed_percision_config/sdxl_turbo/final_config/ac
  --act_protect ./mixed_precision_scripts/mixed_percision_config/sdxl_turbo/final_config/act/
  --res 512 768 1024 \
  --repeats 30 \
  --out ./logs/latency_w6a8.csv


# W4A8
python scripts/latency_probe.py \
  --mode w4a8 \
  --base_path ./logs/sdxl_mixdq_eval \
  --config_weight_mp ./mixed_precision_scripts/mixed_percision_config/sdxl_turbo/final_config
  --config_act_mp ./mixed_precision_scripts/mixed_percision_config/sdxl_turbo/final_config/ac
  --act_protect ./mixed_precision_scripts/mixed_percision_config/sdxl_turbo/final_config/act/
  --res 512 768 1024 \
  --repeats 30 \
  --out ./logs/latency_w4a8.csv
```

Queue tests (static vs adaptative, poisson) FP16

```
[BURST FP16 conc=1] STATIC p50=7479.8ms p95=7520.7ms  |  ADAPTIVE p50=7450.0ms p95=7528.1ms
```

```
[BURST FP16 conc=2] STATIC p50=7673.0ms p95=8690.1ms  |  ADAPTIVE p50=7214.1ms p95=7767.2ms
[POISSON FP16 =0.5 req/s] STATIC p50=7518.8ms p95=7701.2ms  |  ADAPTIVE p50=7270.2ms p95=7423
[POISSON FP16 =1.0 req/s] STATIC p50=7576.0ms p95=7770.7ms  |  ADAPTIVE p50=7252.4ms p95=7349
[POISSON FP16 =2.0 req/s] STATIC p50=7556.7ms p95=7737.5ms  |  ADAPTIVE p50=7217.1ms p95=7345
[POISSON FP16 =4.0 req/s] STATIC p50=7577.2ms p95=7754.4ms  |  ADAPTIVE p50=7298.9ms p95=7668
```

You can have as much text here as you want. The main body must be at most 8 pages long. For the final version, one more page can be added. If you want, you can use an appendix like this one.

The `\onecolumn` command above can be kept in place if you prefer a one-column appendix, or can be removed if you prefer a two-column appendix. Apart from this possible change, the style (font size, spacing, margins, page numbering, etc.) should be kept the same as the main body.