QWEN3.8-FLASH-NEXT · HOMELAB

Paging the n-gram table

By Nate MacFadden · August 2026

Human written (AI code/figures)

This post inspired me to play with Qwen3.8-Flash-Next on my Strix Halo. It made the prospect of putting this model on my card seem easy, in contrast to the fighting/debugging that I normally anticipate. After reading that post, my original goal was to see if I could, offline, serve such a powerful model and push its performance (tokens/sec) higher. This exploration instead ended up being a fun lesson in modern memory-optimization methods which, since so much is going on in ML right now, I was previously unaware of.

The TLDR is that there is a ~30GB table (the n-gram table) which gets ~no memory traffic so the vast majority of it can be offloaded with little hit to decode rate. Of course this isn't a novel observation, but it's fun to see how hard you can push these things. I give below the final config (needs a custom fork/quant... see below) that I used for decode rates near the max, while offloading the n-gram table to disk:

```
llama-server \
  -m Qwen3.8-Flash-Next-UD-IQ4_XS-plesplit.gguf \
  -md Qwen3.8-Flash-Next-MTP-Q8_0.gguf \
  --spec-type draft-mtp --spec-draft-n-max 4 \
  -ngl 999 -fa 1 -b 8192 -ub 2048 -c 131072 --jinja \
  -lm mmap --numa distribute -ot "ple_ngram_embd=CPU"
```

decoding the commands because they still sometimes look foreign to me. First,

```
-md Qwen3.8-Flash-Next-MTP-Q8_0.gguf \
--spec-type draft-mtp --spec-draft-n-max 4
```

enables speculative decoding (have a bit of the model predict future tokens so you can check them in parallel... serial -> parallel enables a speedup even if some tokens are wrong...). Second,

```
-ngl 999 -fa 1 -b 8192 -ub 2048 -c 131072 --jinja
```

are somewhat standard flags.. the `-ngl 999` says to put all layers into VRAM (which we then screw with in the next set of flags), `-fa 1` turns on flash attention, `-b 8192` and `-ub 2048` are batch flags (primarily for pre-fill), `-c 131072` sets the model at a reasonable context of 128k, and `--jinja` enables the Jinja template so the model can use reasoning. Finally,

```
-lm mmap --numa distribute -ot "ple_ngram_embd=CPU"
```

offloads the PLE/n-gram table to the CPU, enabling the big memory reduction. The `-lm mmap` is what makes that work: the table stays a view into the file on disk rather than being copied into a GPU buffer.

To run the above code (at least as of writing this), you need to use the llama.cpp fork, branch vulkan/qwen4exp-rocmfpx, commit 799bf4dd8 (2026-08-27). Also you need to split off PLE heads from the model weights. That can be done via

```
cd llama.cpp        # the fork you cloned above
uv run --no-project --with numpy --with pyyaml \
  python gguf-py/gguf/scripts/gguf_split_ple_heads.py \
  Qwen3.8-Flash-Next-UD-IQ4_XS-00001-of-00003.gguf \
  Qwen3.8-Flash-Next-UD-IQ4_XS-plesplit.gguf
```

Let me know if you find better configs!

Setup

I tried to follow the previous reddit post since it seemed nice, so I used their recommended quant UD-IQ4_XS. I think this is pretty good quality while also fitting on my machine... total (without KV) this should be ~90GB or so. With context it should still be under 128GB.

I also used this provided MTP for speculative decoding since that's generally a free speedup (other than a bit of memory, but the n-gram table turns out super-useful here as I discuss below).

Token Rates

I began trying to get lucky (or persistent) in fiddling with config so as to further optimize the token rates. I had the naive ambition that I could find some nice parameters that led to crazy rates. Mostly struck out there (unless I further quantized weights which made me nervous about token quality), but I did recreate many of the nice rates in this post and others: (measured by reading prompt_per_second and predicted_per_second from llama-server, over 5 wikitext prompts with ignore_eos. These are end-to-end server timings, so they read lower than `llama-bench` on the same box — pp512 is 453 under `llama-bench` here, against the 390 in the linked post.)

TEST	CHANGES TO PARAMETERS	REQUEST	TOKEN RATE [TOKENS/SEC]
prefill, 512-token prompt	none	512-token prompt	287.11 ± 5.93
prefill, 4096-token prompt — default batching	remove `-b 8192 -ub 2048`	4096-token prompt	296.63 ± 2.86
prefill, 4096-token prompt — tuned batching	none	4096-token prompt	357.70 ± 6.24
decode, 128 tokens	remove `-md`, `--spec-type`, `--spec-draft-n-max`	512-token prompt, `n_predict 128`	21.67 ± 0.17
decode, 128 tokens with 16k already in context	remove `-md`, `--spec-type`, `--spec-draft-n-max`	16384-token prompt, `n_predict 128`	16.87 ± 0.15
decode, with multi-token prediction (draft depth 4)	none	512-token prompt, `n_predict 128`	22.04 ± 1.70

Those rates are already nice, but they weren't really budging any further. With that door closed, I looked to the other: maybe the underlying code is inefficient and profiling would pinpoint it. Good and bad news. Good news: most of the time was in memory traffic and matrix multiplies, so there are no obvious algorithmic inefficiencies slowing down my inference. Bad news: there are no obvious inefficiencies. Well, there was an inefficiency in my memory usage due to how *I* configured things...

The n-gram Table

Maybe I'm alone in this but, honestly, I had no idea what an n-gram table or PLE was before this. But it quickly caught my attention when I learned it was taking ~30% of the memory associated to this model (~29GB). It's a big fella, 320,001,446 x 160 of data type IQ4_NL. What surprised me about this was that, despite it costing so much memory, we barely read from it. Initial memory traffic measurements showed it near 0% of the traffic... This is because, each decode step, one just reads only 16 rows, and certain rows tend to be much hotter than others (I'm thinking Zipf style analysis).

This became both a thorn and a curiousity. A thorn because 29GB is a ton of memory, but also a curiousity in how to optimize the data access. I tried the (obvious?) next step: since so little of this table needs to be used in practice, maybe we could use a Zipf-inspired truncation of the table. I.e., store in memory only the most heavily-used rows, defaulting to a disk-read if the model requested an uncommon row. This was inspired by the reduced-vocabulary idea I learned about in my speculative decoding experiments. Of course this is not novel, but learning is still fun :). Such an idea works best if the actively-used table is heavily concentrated and if the tail is thin, which ended up being the case-ish. For a given subject, the hot rows indeed are concentrated and the tail is thin:

Code

CORPUS	TOKENS	ROWS TOUCHED	% OF TABLE	SIZE
othercode	59,675	363,333	0.114%	32.7 MB
config	1,378,402	496,938	0.155%	44.7 MB
shell	276,625	803,108	0.251%	72.3 MB
cccode	464,966	1,475,342	0.461%	132.8 MB
cpp	685,326	1,545,966	0.483%	139.1 MB
python	428,207	1,983,220	0.620%	178.5 MB
all code together	3,293,201	6,162,807	1.926%	554.7 MB

Docs

CORPUS	TOKENS	ROWS TOUCHED	% OF TABLE	SIZE
arxiv_tex	1,040	13,136	0.004%	1.2 MB
linux_docs	11,090	136,054	0.043%	12.2 MB
markdown	820,424	4,578,272	1.431%	412.0 MB
all docs together	832,554	4,692,417	1.466%	422.3 MB

Prose

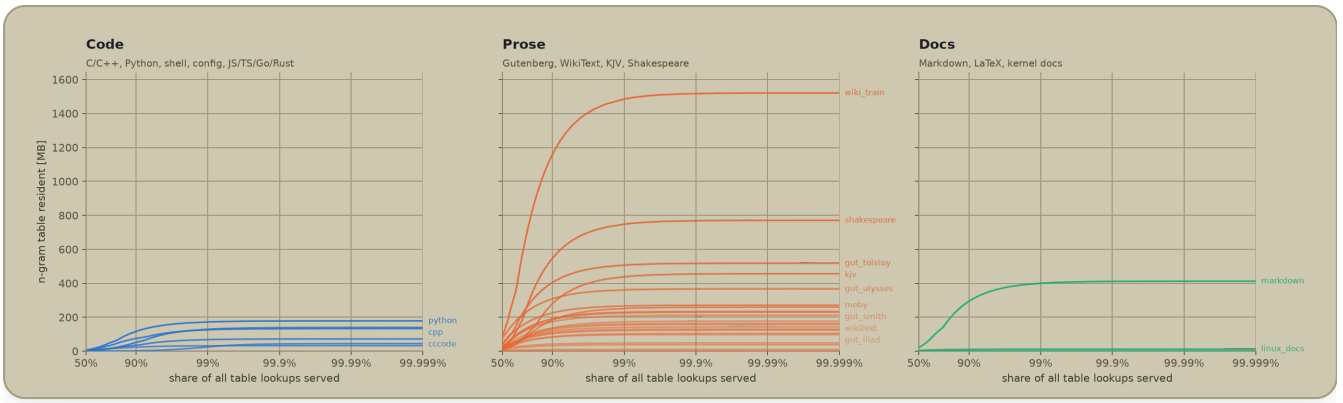
CORPUS	TOKENS	ROWS TOUCHED	% OF TABLE	SIZE
euclid	2,115	18,976	0.006%	1.7 MB
uscode	10,512	119,623	0.037%	10.8 MB
gut_alice	42,133	428,427	0.134%	38.6 MB
gutenberg_scifi	47,565	540,211	0.169%	48.6 MB
gut_frankenstein	103,310	1,093,908	0.342%	98.5 MB
gut_grimm	138,679	1,161,425	0.363%	104.5 MB
gut_doyle	144,634	1,373,744	0.429%	123.6 MB
gut_verne	152,327	1,440,128	0.450%	129.6 MB
austen	179,038	1,582,196	0.494%	142.4 MB
gut_darwin	215,256	1,749,926	0.547%	157.5 MB
gut_dickens	194,348	1,815,751	0.567%	163.4 MB
gut_kant	281,143	1,963,858	0.614%	176.7 MB
wiki_valid	261,411	2,349,379	0.734%	211.4 MB
gut_iliad	297,247	2,546,770	0.796%	229.2 MB
wikitext	297,193	2,603,195	0.813%	234.3 MB
gut_smith	530,173	2,883,747	0.901%	259.5 MB
moby	311,768	3,007,694	0.940%	270.7 MB
gut_ulysses	405,855	4,079,225	1.275%	367.1 MB
kjv	1,224,951	5,070,442	1.585%	456.3 MB
gut_tolstoy	787,268	5,760,175	1.800%	518.4 MB
shakespeare	1,541,993	8,560,882	2.675%	770.5 MB
wiki_train	2,505,800	16,901,205	5.282%	1,521.1 MB
all prose together	9,674,719	47,974,805	14.992%	4,317.7 MB

Everything at once

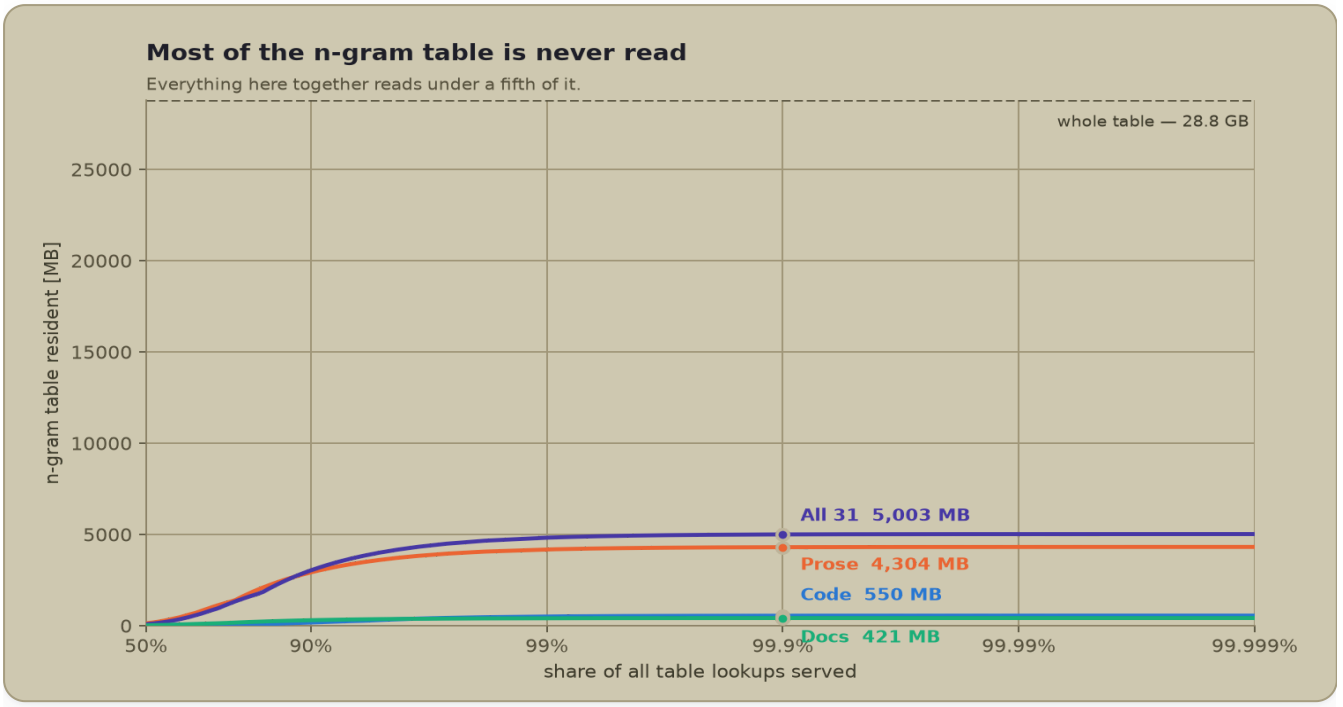
CORPUS	TOKENS	ROWS TOUCHED	% OF TABLE	SIZE
every corpus above	13,800,474	55,800,918	17.438%	5,022.1 MB

One caveat: I fed each corpus whatever text I had, so they aren't the same size. Rows touched grows with tokens, so the sort partly ranks corpus size rather than how concentrated a subject is.

See the figure below:



More aggressively, we can group the subjects and see how much of the table was needed to run over any of these subjects (so run over all corpi corresponding to code, for example).



But the hot-rows don't transfer too well across subjects... E.g., when running over the prose corpi, only 18.8% of reads landed on rows that were also read by the code corpi... That's pretty bad for a single-reduced table idea (also it suggests that Qwen didn't make an unnecessarily large table!) but maybe that's not needed. An LLM session is typically fairly localized (e.g., I don't shift from code to Moby Dick), so a cache (e.g., LRU) might still allow good reductions. Maybe the table could still be aggressively reduced and the (hopefully rare cache misses) would just cost a semi-negligible slowdown compared to the memory saving.

I tried exactly this tradeoff via:

```
-lm mmap --numa distribute -ot "ple_ngram_embd=CPU"
```

This offloads the table to CPU and lets the kernel track it page-by-page (4KB each, though readahead pulls in more like 256KB per fault). Nothing limits how much stays resident other than memory pressure, at which point the kernel drops the pages it hasn't touched recently. To see the cost on performance, I ran those configs over the following corpi (C, C++, Python, shell, Markdown, WikiText, KJV, Shakespeare, Alice, Darwin, Frankenstein, sci-fi, Homer, Kant, Tolstoy, Joyce), same prompts, same order, capped at 192 tokens each (they don't all hit the cap, and I dropped 2 of the 20 prompts that degenerated):

	MEAN RATE	MEDIAN	BEST	SD	ACCEPTANCE	PEAK RAM
table resident	25.45	24.61	45.39	6.76	0.441	114,136 MB
table paged	23.61	22.05	45.89	7.91	0.452	**84,147 MB**

so that's a pretty small hit to decode rate, well within noise, but we reduce the memory footprint by ~30GB! I could probably fit a larger quant because of this.

Sizes of Qwen3.8-Flash-Next in the Limit of No Table

So I got curious about running this on my other computer (which has ~48GB VRAM, ~30GB system RAM). This is a more traditional machine with a CPU, RAM sticks, and discrete GPU. It ended up not working (yet... well I got a trickle of ~12 tok/sec) but I did record below, in the limit of no n-gram table, what I could run:

QUANT	TOTAL	TABLE	EVERYTHING ELSE	FITS W/ KV ON 47.8 GIB?
UD-IQ1_S	67.6 GiB	26.8	40.7	yes
UD-IQ1_M	69.4 GiB	26.8	42.6	barely
UD-Q2_K_XL	73.5 GiB	26.8	46.6	no
UD-IQ3_XXS	76.3 GiB	26.8	49.5	no
UD-Q3_K_XL	83.8 GiB	26.8	57.0	no
UD-IQ4_XS	87.2 GiB	26.8	60.4	no
UD-Q4_K_XL	103.7 GiB	26.8	76.9	no