Why a smaller model writes faster, *from the memory chip upward*

Text generation on a phone is paced mostly by memory traffic, so smaller weights usually mean faster replies. Published measurements show where that proportional rule bends.

MITWA LAB, MUMBAI

IN PLAIN WORDS

A language model writes a reply one small piece of text at a time, and for every piece it has to read its whole store of learned numbers out of the phone's memory. On a phone the processor can usually calculate faster than memory can hand it those numbers, so the reading sets the pace, and a model half the size should write about twice as fast. In the published measurements reviewed here, that rule works as a ceiling. Shrinking a model's file has given less speed-up than the shrinkage, and in some tests the smaller file ran slower, in one case four times slower than the uncompressed model. Reading the question also behaves differently from writing the answer, and in one test a phone that grew warm lost about 40% of its speed within minutes.

The memory bus sets the ceiling

Any processor can only work on data that has arrived from memory. Williams et al. (2008) turned this into a simple bound, the roofline: the work a program can do per second is the lower of two ceilings, the processor's peak arithmetic rate and the memory bandwidth multiplied by the number of operations the program performs on each byte it fetches from main memory. A program that does little arithmetic per byte hits the second ceiling and is called memory-bound; one that does a lot hits the first and is compute-bound. The authors wrote in 2008 that they expected off-chip bandwidth would often be the binding limit for the foreseeable future.

Since then the gap has widened, and Gholami et al. (2024) estimate that over twenty years the peak arithmetic of server hardware grew about 60,000-fold while DRAM bandwidth grew about 100-fold. Those figures describe data-centre chips. For phones the published figures are thinner. Apple's announcement of the A18 Pro, the chip in the iPhone 16 Pro, claims 17 percent more memory bandwidth for the whole system than the chip before it, and gives no absolute number (Apple, 2024). Phones also share one memory between all their processors, and Huang et al. (2025) point out that during text generation every core waits on the same bus, which makes generation hard to speed up at the level of the whole system.

All of this assumes the model fits in working memory. When it does not, the weights must come from flash storage, which is slower than DRAM in both latency and throughput; Alizadeh et al. (2024) report more than 6 GiB a second for a 1 GiB sequential read on a Mac with an M1 Max, less for small scattered reads, and note that a naive design reloading the whole model on every step would take seconds per step. With careful loading they ran models as large as twice the DRAM on the machine. The rest of this review is about models that fit, which is where speed follows bytes.

What one token asks of the processor

A model produces text in tokens, pieces of a word or short whole words. Pope et al. (2022) give the two costs of producing one. The arithmetic is about two operations per weight, a multiplication and an addition, so a model with N weights needs about 2N operations per token. Separately, the weights and the stored attention state (the cached results for earlier tokens, which the field calls the KV cache) must travel

from memory to the processor once per step. Pope and colleagues call these the compute time and the memory time, and note that when few requests are served together and sequences are short, loading the weights dominates.

Loading dominates because, when one request is served at a time, generating a token multiplies each weight matrix by a single vector, so every weight fetched serves that one token and the cost of fetching it cannot be spread across others (Kim et al., 2024; Huang et al., 2025). That is very little arithmetic per byte, which places generation on the bandwidth side of the roofline. Gholami et al. (2024) show the effect on an Intel CPU at batch size 1, where GPT-2 ran slower than BERT-Base although the two have largely the same configuration and total operation count, because GPT-2 generates one token at a time. On the small models they profiled with llama.cpp, Lu et al. (2024) found matrix-by-vector multiplication took over 70% of end-to-end time.

If reading the weights is the whole cost, time per token is the weight file's size divided by bandwidth, which gives the proportional rule that a model k times smaller is k times faster. A clear statement of the claim comes from Kim et al. (2024), whose roofline-based performance model of LLaMA-7B on an A5000 GPU showed latency falling linearly as the bits per weight fell. That is a model of the hardware, and their own measurements on a different GPU (an A6000), described below, showed a speed-up well short of the fall in bits. Our technical note 03 used this first-order bound for Bloom's two builds and said in its limitations that the true gain may be smaller than the ratio of their sizes; that note reported no measured speed for Bloom, and nothing in this review is a measurement of Bloom.

The measured gain is smaller than the shrinkage

Apple's machine learning research group ran Llama-3.1-8B-Instruct on the GPU of a Mac with an M1 Max, a processor it chose because such models are usually limited by memory bandwidth (Apple Machine Learning Research, 2024). Storing the weights in 4 bits instead of 16 cut the model from 16 GB to 4.2 GB, about 3.8 times smaller, and raised generation from 16.26 to about 33.7 tokens a second, about twice as fast. By our arithmetic, if each byte of the weights were read once per token, the larger build would be moving about 260 GB a second and the smaller about 141 GB a second, which suggests that in the smaller build something other than reading weights takes a larger share of each token.

Data-centre hardware shows the same pattern, though in a different regime from serving one request at a time. For PaLM 540B at batch size 64 on TPU v4 chips, halving the bytes per weight cut the time per token from 36.9 to 28.5 milliseconds, about 1.3 times faster (Pope et al., 2022).

Kim et al. (2024) measured both a gain and a loss. They generated 128 tokens with LLaMA-7B on an A6000 GPU in 3.2 seconds at 16 bits and in 1.4 seconds with a 3-bit version (peak memory 12.7 GB against 2.9 GB), a 2.3-fold speed-up for a more than fourfold cut in memory. A grouped 3.25-bit version with a particular reordering took 13.7 seconds, more than four times slower than the 16-bit model while using 3.0 GB, which the authors attribute to scattered memory access. Two independent measurement studies report that 3-bit weights ran slower than 4-bit ones. Lu et al. (2024), testing one small model, Phi-1.5, with llama.cpp, attribute it to the irregular width lacking hardware support, and Laskaridis et al. (2024), across phones and Jetson boards, found 4-bit models about 25% faster on average. Fewer bytes pay off only to the extent that the hardware and software can unpack them cheaply.

Lu et al. (2024) also report a case where the model with more weights was the faster one. Qwen1.5-0.5B, with 25.4% more weights than Qwen2-0.5B, ran 31.9% faster on a Jetson Orin NX, and the authors caution that this is likely hardware-dependent. The difference sat mainly in processing the input; when generating, the two came close, which fits their finding that generation follows size more nearly linearly than input processing does.

Reading the question and writing the answer

A reply is produced in two phases. In the first, often called prefill, the model reads the whole input, and because every input token is present at the start, they can be processed together in parallel (Pope et al., 2022). In the second, decoding, it writes the output one token at a time, each step waiting for the last. Prefill does more arithmetic per byte fetched and leans towards the compute ceiling, while decoding is memory-bound (Lu et al., 2024; Huang et al., 2025). On a Xiaomi 15 Pro running on its CPU, decoding was three times slower per token than prefill (Huang et al., 2025).

The phases respond differently to hardware and to size. On an 11-inch iPad Pro with Apple's M1, moving to the GPU made prefill three to four times faster for every model tested, while decoding ran at about the CPU's speed (Xiao et al., 2024); the same study found

decoding speed varied little across top-tier devices, as a memory-bound task should. Lu et al. (2024) found that lower-precision weights cut decoding latency by between 17% and 75% for one small model, while their benefit to prefill shrank as the input grew longer, because the weights are fetched once for the whole input. As inputs grow longer, then, a smaller file helps mainly with the steady rate at which words appear and less with the wait before the first one. With very short inputs the distinction blurs. For a 7-token input, Apple's model produced its first token in 128.32 milliseconds with 16-bit weights and in 51.91 with 4-bit ones (Apple Machine Learning Research, 2024). In the conversational datasets Huang et al. (2025) examined, the output ran about 3.5 times longer than the input, which together with its slower rate made decoding the longer phase in their measurements.

A whole reply depends on more than the weights

Apple's 4-bit change was made to the weights, and the stored attention state, which grows with each token and is read at every step, is a separate cost whose effect shows in Apple's tables. As the maximum length the model was prepared for rose from 128 to 8,192 tokens, the 16-bit build slowed from 17.89 to 12.31 tokens a second and the 4-bit build from 39.44 to 21.46; Apple attributes part of the slowdown to computations that scale with that length and to the cache outgrowing the GPU's own cache beyond 2,048. By our arithmetic the 4-bit build's advantage fell from about 2.2 times to about 1.7 times across that range (Apple Machine Learning Research, 2024).

Software choices can move speed by more than size does. In the same Apple post, the first 16-bit export, with no cache for earlier tokens and inputs padded to full length, ran at 0.19 tokens a second, and the version holding that cache as model state ran at 16.26, with the weights unchanged. Laskaridis et al. (2024) found one inference engine generating up to 3.53 times faster than another, though on average the gap was 4% for GPU execution and 28% against the other engine running on the CPU. Huang et al. (2025) report their own engine 12 to 363% faster on average than four others across five Android and two iOS devices, a comparison by the engine's authors. Leviathan et al. (2023) changed speed without changing the model at all. Because generation leaves arithmetic idle while it waits on memory, a small helper model can propose several tokens that the large model checks in one pass, and on T5-XXL at batch size 1 on a TPU v4 they measured speed-ups of 2.3 to 3.4 times with outputs distributed exactly as before.

Heat can play a part even within a single reply. On a Meizu 18 Pro, Lu et al. (2024) saw the time per token climb steeply over the first ten generated tokens and then level off, which they attribute to the chip lowering its clock as it warmed.

Ten minutes of continuous use

Tummalapalli et al. (2026) ran a 4-bit version of Qwen 2.5 1.5B on an iPhone 16 Pro's GPU through the MLX Swift library, generating exactly 819 tokens from a fixed 258-token input, twenty times with a one-second pause. In the first two runs the phone peaked at 40.49 tokens a second; from the third it reported itself warm and averaged 24.96; from the seventeenth it was hot and held at 23.67, 41.5% below peak. By our arithmetic the whole test lasted about eleven minutes. The study has clear limits, most of them stated by its authors: power could not be measured on the iPhone, one model was tested with one long input, and the results describe hardware and software together. It is also a company's preprint. An older study points the same way. On an iPhone 14 Pro, a 4-bit version of Zephyr-3B run on fifty requests in a row lost speed from the start, with further drops around the twentieth and thirty-second requests (Laskaridis et al., 2024).

What this adds to the proportional rule is a time axis, since a speed ratio measured on a cool device may not hold once the device is hot. Whether two sizes of the same model keep their ratio once both are throttled is a question none of the studies reviewed here answers.

What is not yet known

- Apple's announcement of the A18 Pro gives only a relative bandwidth figure, and no absolute figure for any iPhone was found in the sources reviewed, so a bound in milliseconds per token for a given phone rests on an assumed bandwidth.
- No study found here measures two sizes of one model, on one iPhone and one software stack, cold and under sustained heat, which is the comparison that would show whether the size advantage survives throttling.
- How the gap between the proportional rule and measured speed divides between unpacking compressed weights, reading the stored attention state and software overhead has not been separated on a phone in any study reviewed here; Apple's worked example ran on a Mac.

- The iPhone 16 Pro thermal curve rests on one company preprint with one model and one input, and no independent repeat on the same phone was found in the sources reviewed here.
- The sustained-load study could not measure power on the iPhone 16 Pro, so whether a smaller model's energy saving on that phone keeps pace with its speed gain under continuous use is open.

References

Alizadeh, K., Mirzadeh, I., Belenko, D., Khatamifard, S. K., Cho, M., Del Mundo, C. C., Rastegari, M., and Farajtabar, M. (2024). LLM in a flash: Efficient large language model inference with limited memory. In Proceedings of ACL 2024. arxiv.org/abs/2312.11514

Apple (2024). Apple debuts iPhone 16 Pro and iPhone 16 Pro Max. Apple Newsroom, 9 September 2024. www.apple.com/newsroom/2024/09/apple-debuts-iphone-16-pro-and-iphone-16-pro-max/

Apple Machine Learning Research (2024). On device Llama 3.1 with Core ML. 1 November 2024. machinelearning.apple.com/research/core-ml-on-device-llama

Gholami, A., Yao, Z., Kim, S., Hooper, C., Mahoney, M. W., and Keutzer, K. (2024). AI and memory wall. IEEE Micro; extended version arXiv:2403.14123. arxiv.org/abs/2403.14123

Huang, Z., Niu, C., Wang, Z., Xue, J., Zhang, H., Wang, Y., Xin, Z., Jiang, X., Lv, C., Wu, F., and Chen, G. (2025). MNN-AECS: Energy optimization for LLM decoding on mobile devices via adaptive core selection. arXiv:2506.19884. arxiv.org/abs/2506.19884

Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M. W., and Keutzer, K. (2024). SqueezeLLM: Dense-and-sparse quantization. In Proceedings of ICML 2024. arxiv.org/abs/2306.07629

Laskaridis, S., Katevas, K., Minto, L., and Haddadi, H. (2024). MELTing point: Mobile evaluation of language transformers. In Proceedings of MobiCom 2024. arxiv.org/abs/2403.12844

Leviathan, Y., Kalman, M., and Matias, Y. (2023). Fast inference from transformers via speculative decoding. In Proceedings of ICML 2023, PMLR 202. arxiv.org/abs/2211.17192

Lu, Z., Li, X., Cai, D., Yi, R., Liu, F., Zhang, X., Lane, N. D., and Xu, M. (2024). Small language models: Survey, measurements, and insights. arXiv:2409.15790. arxiv.org/abs/2409.15790

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. (2022). Efficiently scaling transformer inference. arXiv:2211.05102. arxiv.org/abs/2211.05102

Tummalapalli, P., Arayakandy, S., Pal, R., and Kundan, K. (2026). LLM inference at the edge: Mobile, NPU, and GPU performance efficiency trade-offs under sustained load. arXiv:2603.23640. arxiv.org/abs/2603.23640

Williams, S., Waterman, A., and Patterson, D. (2008). Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, University of California, Berkeley. www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.pdf

Xiao, J., Huang, Q., Chen, X., and Tian, C. (2024). Understanding large language models in your pockets: Performance study on COTS mobile devices. arXiv:2410.03613. arxiv.org/abs/2410.03613