ROGER'S
RABBIT HOLE



A ROGER'S RABBIT HOLE WHITE PAPER

MIXTURE-OF-EXPERTS MODELS

Breaking the relationship between model size, computational demand and local AI hardware



BY ROGGIE D

NOVEMBER 2026

rogersrabbithole.com

MIXTURE-OF-EXPERTS MODELS

Breaking the relationship between model size, computational demand and local AI hardware

By Roggie D | November 2026

WHO THIS IS FOR

Local AI hobbyists, home lab builders, IT people and developers who keep seeing model names like 35B-A3B and want to know what the second number means for the hardware they already own.

HOW I LABEL THE NUMBERS

CyberTron numbers in this paper come from my own project records: benchmark files from Oct. 16 and Oct. 21, 2026, and observations from November. I didn't run new benchmarks for this paper. Numbers from research papers and model pages are those authors' figures, not mine. Calculations are labeled as calculations, and illustrations are labeled as illustrations.

Related reading on Roger's Rabbit Hole: Bigger Isn't Always Smarter, Building AI on Aging Hardware, GPU VRAM: The Real Currency of Local AI and Local AI Model Racks, all at rogersrabbithole.com.

THE SHORT VERSION

For most of the short history of large language models, the rule was simple. A bigger model needs a bigger computer. More parameters meant more GPU memory, more system memory, more power and more money.

Mixture-of-experts models bend that rule. Instead of running the whole network for every word, a mixture-of-experts model, or MoE, uses a small learned router to pick a few expert subnetworks for each token. The model can hold a very large total number of parameters while using only a fraction of them for any one token.

The model on my everyday slot shows both sides of that trade. It's a community build based on Qwen3.6-35B-A3B. That name means about 35 billion parameters in total and about 3 billion active per token. So each token does roughly the math of a 3B model. But

all 35 billion parameters still have to live somewhere, and the file is around 22 GB. My graphics card has 12.

That's the key idea of this paper: sparse computation is not sparse storage. MoE cuts the work per token. It doesn't cut the memory you need to hold the model. On an old AMD FX-8300 with 32 GB of DDR3 and a 12 GB RTX 3060, the practical way to run a model like that at 4-bit is to split it, keeping the busiest parts on the GPU and parking many of the expert weights in system RAM.

The first time I tried, on the night of Oct. 15 into Oct. 16, my first clean benchmark averaged about 5.57 tokens per second. With better placement that night, I reached 11.75. Three days later, a real session on my early, untuned setup generated 4.82 tokens per second. On Oct. 21, with every layer on the GPU except the expert weights of 26 layers, the same kind of benchmark showed about 18. On the build and settings I use now, completed answers run around 19 tokens per second.

> **MoE doesn't remove hardware limits. It changes which limit you hit first, and smart placement decides how much of the model's promise you actually get.**

1. THE BRUTE-FORCE PROBLEM

The industry sells model capability by parameter count: 7 billion, 70 billion, hundreds of billions, a trillion. The implied message is that more parameters make a smarter model, and that more hardware is the natural fix.

That message shuts out a lot of people. A dense 70B model needs tens of gigabytes of memory even after compression. The usual answer is the one every infrastructure engineer knows: buy more GPUs, more memory, more power and more cooling.

MoE asks a different question. How much of the network actually needs to take part in any one computation? That's a shift from brute-force provisioning toward smarter use of the resources you have.

I've argued in earlier papers that bigger isn't automatically smarter, that an old PC can run real local AI if you respect its limits, and that VRAM is the budget everything else spends. This paper looks inside the model itself, at an architecture that was built to get more capacity out of less computation.

2. DENSE MODELS: EVERY TOKEN TAKES THE SAME ROAD

A traditional dense language model runs the same learned components for every token that passes through its transformer layers. The numbers flowing through change from token to token, but the path through the parameters stays the same.

Dense models are still very useful. Their execution is predictable, they're simple to place in memory, and the software for them is mature. A small dense model that fits entirely on the GPU is often the fastest thing you can run on a consumer card. The catch is that capacity and work move together. Make a dense model bigger, and every token costs more.

3. WHAT A MIXTURE-OF-EXPERTS MODEL IS

An MoE model contains trainable subnetworks called experts and a learned routing mechanism, often called a router or gate. In modern sparse MoE transformers, some or all of the feed-forward blocks are replaced with a set of expert networks. For each token, the router scores the experts and picks a small number of them. Only those experts run, and their outputs are blended using the router's weights.

The idea isn't new. In 2017, Noam Shazeer and colleagues at Google Brain introduced a sparsely gated MoE layer with up to thousands of expert subnetworks and reported more than 1,000 times the model capacity with only minor losses in computational efficiency [1]. In 2021, Google's Switch Transformer simplified routing to a single expert per token and trained sparse models up to a trillion parameters [2]. Mistral AI's Mixtral 8x7B, released with open weights, uses eight experts per layer and picks two per token, for 47 billion parameters in total and 13 billion active [3].

Some newer designs add shared experts that run for every token next to the routed experts. DeepSeek described isolating shared experts to capture common knowledge and reduce redundancy among the routed ones [4]. Qwen3.6-35B-A3B lists 256 experts, with 8 routed experts plus 1 shared expert active for each token [5].

COMPONENT	WHAT IT DOES
Router (gate)	Scores the experts for each token and picks the top few
Routed experts	Run only when the router selects them
Shared experts	Run for every token, where the design includes them
Attention	Relates each token to the others in the context
Aggregation	Blends the selected experts' outputs using the router's weights
Load balancing during training	Encourages the router to use all the experts, not just a favorite few

Table 1. The parts of an MoE layer

A networking analogy helps, up to a point. A dense model is like a fixed processing pipeline. An MoE model is more like policy-based traffic steering. But the router's policy is learned during training, it works on numbers that represent tokens, and its experts aren't neat departments with job titles. When Mistral's researchers looked at which experts Mixtral picked for ArXiv papers, biology abstracts and philosophy papers, they found no obvious pattern by topic. Expert choice seemed more aligned with syntax than with subject [3]. Experts aren't little AI employees with job descriptions.

4. THE MATH BEHIND TOTAL AND ACTIVE PARAMETERS

For an MoE model with equal-sized routed experts, the totals work out like this:

- Total parameters = shared parameters + (number of experts x parameters per expert)
- Active parameters, approximately = shared parameters + (experts selected x parameters per expert)
- Active fraction = active parameters / total parameters

For a 35B-A3B model, the active fraction is 3 / 35, or about 8.57 percent (calculated). Put another way, the total is about 11.7 times the active count. That isn't a speedup estimate. It's a ratio of parameter counts, and real speed depends on memory, placement and software, as the rest of this paper shows.

At the expert level, Qwen3.6-35B-A3B's router picks 8 of 256 routed experts, or 3.125 percent (calculated). Add the shared expert and 9 of 257 experts run, about 3.5 percent. Neither number matches the 8.57 percent overall figure, because attention, embeddings, the router and other shared parts also run for every token.

MODEL	TOTAL PARAMETERS	ACTIVE PER TOKEN	EXPERTS (SELECTED)	ACTIVE FRACTION (CALCULATED)
Any dense model	All	All	None	100%
Mixtral 8x7B [3]	47B	13B	8 (2)	About 28%
Qwen3-30B-A3B [6]	30.5B	3.3B	128 (8)	About 11%
Qwen3.6-35B-A3B [5]	35B	3B	256 (8 routed + 1 shared)	About 8.6%

Table 2. Total and active parameters, from published model specifications

A 35B MoE holds nearly four times the parameters of a dense 9B model while activating about a third as many per token. Whether that turns into better answers or faster ones is an experimental question, not something the parameter counts can settle.

5. THE MEMORY MISCONCEPTION

Here's the part the hype skips. Sparse computation is not sparse storage. Every one of those 35 billion parameters has to stay reachable, because the router can pick any expert for the next token. They can sit in VRAM, in system RAM, in memory-mapped storage or in a mix of all three, but they have to be somewhere.

The minimum storage for the weights alone is the parameter count times the bits per weight, divided by eight.

PRECISION	WEIGHT STORAGE
32-bit	140 GB
16-bit	70 GB
8-bit	35 GB
6-bit	26.25 GB
5-bit	21.875 GB
4-bit	17.5 GB
3-bit	13.125 GB
2-bit	8.75 GB

Table 3. Theoretical weight storage for a 35-billion-parameter model (calculated)

Hugging Face's MoE explainer makes the same point: all the parameters have to be loaded, so memory requirements are high [13]. Even the theoretical 4-bit figure is bigger than a 12 GB card. Real files run bigger still. The October file I tested was about 21.7 GB for about 35.5 billion parameters, which works out to about 4.9 bits per weight on average (calculated from the benchmark tool's own file record). A "4-bit" file isn't 4 bits per weight across the board. The build I run now is about 22.1 GB.

And the weights aren't the whole bill. Runtime memory is the weights plus the attention cache, compute buffers and runtime overhead. Long contexts, several sessions at once, vision encoders, embedding models and anything else on the GPU all add pressure.

6. CYBERTRON: THE CASE STUDY

CyberTron is my project to see how much useful AI I can run on repurposed equipment, a consumer GPU, open-weight models and locally hosted software. Rather than building an expensive system around a model, the idea is to fit models and their placement to the hardware I have.

COMPONENT	RECORDED CONFIGURATION
CPU	AMD FX-8300, eight integer cores in four modules [19], no AVX2
System memory	32 GB of DDR3, four 8 GB modules running at 1600 MT/s
GPU	NVIDIA RTX 3060 with 12 GB of GDDR6 [9]
GPU link	PCIe x16, limited to PCIe 2.0 by the older board
Runtime	llama.cpp with CUDA

Table 4. CyberTron hardware as recorded for this paper

Building AI on Aging Hardware tells the story of how this box came together [15].

Model provenance. The 35B model in this paper is a community build based on Qwen3.6-35B-A3B, which Qwen publishes under the Apache 2.0 license [5]. A community build isn't necessarily identical to the publisher's own checkpoint. The October file and the file I use now are two different community builds of the same base model. In mid-November I swapped from the first to the second because the first had trouble following formatting instructions. Section 17 covers why the source of a model file matters.

7. HETEROGENEOUS COMPUTING: MAKING THE MODEL FIT

When a model is bigger than VRAM, llama.cpp can split it between the GPU and the CPU. The traditional way is layer by layer: put some number of whole layers on the GPU and leave the rest on the CPU. For MoE models there's a finer tool. llama.cpp can keep the expert weights of a chosen number of layers in system memory, while the rest of each layer can still go on the GPU [7].

That matters because of how MoE works. The attention parts and shared parts run for every token, so they tend to benefit most from the GPU's fast memory. Each expert, by contrast, runs only when it's picked. Parking the experts in system RAM frees VRAM for the parts that never rest.

OPTION	WHAT IT CONTROLS
--n-gpu-layers	How many layers go on the GPU
--cpu-moe	Keep all MoE expert weights on the CPU
--n-cpu-moe N	Keep the MoE expert weights of N layers on the CPU
--ctx-size	How much context memory to allocate
--threads	How many CPU threads to use

Table 5. llama.cpp placement options used in this paper [7]

None of this makes 32 GB of RAM plus 12 GB of VRAM equal to 44 GB of fast GPU memory. Experts that live in system memory are read at system memory speed, and Section 9 shows how big that gap is. Research also warns that expert offloading suits some models better than others. A 2025 study of 20 MoE models found that how consistently a model picks the same experts for neighboring tokens varies from model to model, and that shared experts can lower that consistency [8]. That affects how well any caching of hot experts can work.

8. EVIDENCE FROM CYBERTRON

EXHIBIT A: THE NIGHT PLACEMENT MATTERED

In mid-October a post claimed a 35B mixture-of-experts model ran at about 50 tokens per second on a 12 GB RTX 3060. That's my card. On the night of Oct. 15 into Oct. 16, I built llama.cpp and tried the community build. I wrote up that night in A Post Said 50 Tokens a Second [17]. Here's what it showed about MoE placement.

SETTING	GPU LAYERS	EXPERT WEIGHTS KEPT ON THE CPU	TOKENS PER SECOND
Out of the box, 8 threads, 16-bit cache	16	None	5.57
Flash attention on, 8-bit cache, 4 threads	16	None	7.22
Same, experts moved to CPU	16	8 layers	7.15
Same, more experts moved to CPU	16	26 layers	7.15
More layers on the GPU	24	26 layers	8.92
More layers on the GPU	32	26 layers	11.75

Table 6. Oct. 16, 2026, generation-only benchmarks (no prompt, context size not recorded), same file, same card

Moving experts to the CPU by itself did nothing, 7.15 against 7.22. The gain came when the VRAM that freed up was spent on more GPU layers: 8.92 with 24, then 11.75 with 32. That's the MoE placement story in one table. Offloading experts isn't the win. Using the room it frees is.

The 7.22 row changed three settings at once, so I can't say which one did the work. The first row is the average of three runs of 128, 256 and 512 tokens, at 5.57, 5.63 and 5.49 tokens per second. My earlier write-up gave 5.56, from a run I pasted a few minutes before these. The saved benchmark file, used here, averages 5.566, which rounds to 5.57. The other rows used 256-token runs, each repeated three times. These benchmarks generated text from an empty prompt, so they show placement, not real-world speed. The 11.75 setup never passed a memory check that night, so I didn't make it my everyday setting.

EXHIBIT A, CONTINUED: THE EARLY DEPLOYMENT, OCT. 19

The setup I actually ran first was the 7.22 one: an 8K context, 16 layers on the GPU, 4 threads, flash attention, an 8-bit cache and no expert offload. On Oct. 19, a real request on that setup read a 5,241-token prompt in 34.9 seconds and then generated 512 tokens in 106.0 seconds. The log put that at 4.82 tokens per second. A real request carries its prompt in context and the benchmark didn't, which is one likely reason the session ran slower than the 7.22 benchmark. That was the starting line before tuning. The jump from there to about 19 tokens per second today came from placement and configuration on the same card, not new hardware.

EXHIBIT B: AN MOE AGAINST A DENSE MODEL, OCT. 21

On Oct. 21, CyberTron ran a side-by-side evaluation of three models on the same llama.cpp build, the same 16K context and the same test suite. Two were community builds based on Qwen3.6-35B-A3B, with every layer on the GPU except the expert weights of 26 or 23 layers. The third was a 27-billion-parameter dense model compressed to about 3.4 bits per weight so it could fit entirely on the GPU.

MEASURE	OCTOBER 35B MOE BUILD, EXPERTS OF 26 LAYERS ON THE CPU	LATER 35B MOE BUILD, EXPERTS OF 23 LAYERS ON THE CPU	27B DENSE, FULLY ON THE GPU
Prompt processing, 512 tokens	211 tokens per second	225 tokens per second	546 tokens per second
Generation, 128 tokens	18.2 tokens per second	17.2 tokens per second	20.5 tokens per second
Wait for first token on a prompt of about 9,600 tokens	47.9 seconds	42.4 seconds	19.0 seconds
Peak GPU memory, whole card, including about 0.9 GB held by other programs	9,859 MiB	11,827 MiB	11,843 MiB
Suite score	105 of 110 (no vision test)	111 of 115	110 of 115

Table 7. Oct. 21, 2026, side-by-side on a 12 GB card, 16K context

Three things stand out. First, the MoE builds generated text almost as fast as a dense model with nine times as many active parameters (27B against 3B, calculated), while holding 35 billion parameters in total. Second, the dense model read prompts about 2.4 times as fast (546 / 225, calculated), which cut the wait on a long prompt by more than half. Prompt processing works on many tokens at once, and these results suggest that's where keeping experts in system memory costs the most. Third, the quality scores were within noise of one another. The evaluation's own write-up said a one-point difference was noise, and that most sections were near their ceiling.

Run once a second across the whole suite, the October build's card use stayed between 9,813 and 9,859 MiB, about 9.6 GiB. That's the whole GPU, other programs included, with every layer on the GPU except the expert weights of 26 layers, at a 16K context.

Placement had a hard edge, too. With the later build, keeping the experts of only 22 layers on the CPU failed to load for lack of memory. Twenty-three worked, peaking with about 461 MiB of the card to spare.

EXHIBIT C: HOW IT RUNS NOW

Since Oct. 21, a 35B mixture-of-experts build has handled my everyday requests. Today that's the later community build based on Qwen3.6-35B-A3B, with every layer on the GPU except the expert weights of 26 layers, and a 32K context. The 9B Qwen model from early October is still installed as a smaller, faster option. I describe the setup as an inventory matched to workloads, not one model for everything.

MEASURE	OBSERVATION
Model file	About 22.1 GB
Context	32,768 tokens
Placement	All layers on the GPU except the expert weights of 26 layers
GPU memory held by the model server	9,682 MiB, with 10,798 of 12,288 MiB used on the card
System memory held by the model server	About 14 GB
Generation, completed answers	About 19 tokens per second
Long prompt, about 18,000 tokens	Read at about 264 tokens per second; about 83 seconds to finish

Table 8. The current 35B configuration, November observations

Those are single observations from Nov. 20 and Nov. 23, not a formal benchmark. They aren't a redo of the Oct. 16 or Oct. 21 tests either, because the build, context and settings are different.

WHAT THE EXHIBITS PROVE, AND WHAT THEY DON'T

Fact: a 35B-total MoE model ran on a 12 GB card with an FX-era CPU and DDR3 by splitting expert weights into system memory. On the same hardware, placement alone took generation from about 5.57 to 11.75 tokens per second in benchmarks, and later configurations ran near 18 to 19.

Fact: in one side-by-side, a dense 27B that fit on the GPU read long prompts about 2.4 times as fast as the MoE builds and generated a little faster. Quality was within noise.

Not shown: that an MoE model gives better answers than a dense one, or that the 35B is smarter than my 9B. There's no formal accuracy benchmark of the 35B against the 9B in my records. An early October reasoning test clocked the 9B at about 66.6 tokens per second with thinking off, but that test used a different runtime and suite, and most of those answers were only two tokens long, which flatters the rate. With thinking on, the

same test averaged about 43. Unlike tests aren't a head-to-head, and I haven't measured which experts fire on real prompts.

9. THE MEMORY BANDWIDTH PROBLEM

Capacity and bandwidth are different things. Capacity is how much fits. Bandwidth is how fast you can read it.

PATH	NOMINAL BANDWIDTH
RTX 3060 12 GB GDDR6 [9]	360 GB/s
Dual-channel DDR3-1600 system memory [10] [11]	25.6 GB/s (2 x 12.8 GB/s, calculated)
PCIe 2.0 x16, one direction [12]	About 8 GB/s

Table 9. Nominal bandwidth on CyberTron's paths (published specifications, not my measurements)

The GPU's memory is about 14 times as fast as the system memory on paper (360 / 25.6 = 14.06, calculated). That isn't an inference speed multiplier, but it shows why the parts that run for every token belong on the GPU.

A common rule of thumb is that token generation is limited by memory bandwidth. A simple ceiling is effective bandwidth divided by the bytes read per token. As a rough illustration, if all 3 billion active parameters of my October file, at about 4.9 bits each, had to come from DDR3, that's about 1.8 GB per token and a ceiling near 14 tokens per second. From the GPU's memory alone, the ceiling would be near 200. Those are back-of-the-envelope numbers, not predictions. Caching, batching, routing and compute limits all change the picture.

Splitting a model also means passing data between the CPU and GPU and waiting for each other. Repeated transfers, poor reuse and a slow link add up. One more thing to watch: my idle snapshot showed the GPU link running at first-generation PCIe speed, so measure the link under load, not at rest.

10. CONTEXT LENGTH: THE HIDDEN RESOURCE CONSUMER

Context length is the token history the model can see in one session. Qwen3.6-35B-A3B supports 262,144 tokens natively [5]. My first October setup ran at 8,192. Today I run 32,768, one-eighth of the native figure.

The attention cache grows with context. For a conventional attention layer, the cache is 2 (keys and values) x layers x tokens x key-value heads x head size x bytes per value. Qwen3.6-35B-A3B is a hybrid. According to its model page, only 10 of its 40 layers use full gated attention, with 2 key-value heads of size 256. The other 30 use Gated DeltaNet, a linear-attention design, so the conventional cache formula covers only the 10 attention layers [5]. The metadata in my current file agrees: 40 blocks. The earlier community build's file reported 41, because it counted one extra multi-token prediction layer, which my runtime loaded and then ignored.

CONTEXT	16-BIT CACHE	8-BIT CACHE
8,192 tokens	160 MiB	80 MiB
32,768 tokens	640 MiB	320 MiB
131,072 tokens	2,560 MiB	1,280 MiB
262,144 tokens	5,120 MiB	2,560 MiB

Table 10. Attention cache for the 10 full-attention layers of Qwen3.6-35B-A3B (calculated)

That's the cache for one session and those layers only. The linear-attention state, compute buffers and runtime overhead come on top, so get the real numbers from your runtime's logs. The calculation shows why this model can carry a 32K context on a tight card. With the same head layout in all 40 layers, the cache would be four times as big (calculated).

There's a trade-off to be honest about. Qwen's model page advises keeping at least 128K tokens of context to preserve the model's thinking capabilities [5]. My 32K setting is a hardware compromise. Use the smallest context that meets the job, and remember that context isn't permanent memory and doesn't automatically improve reasoning.

WORKLOAD	STARTING CONTEXT
Basic conversation	4,096 to 8,192
Technical help	8,192 to 16,384
Code analysis	8,192 to 32,768
Retrieval-augmented generation	8,192 to 16,384
Long document analysis	32,768 or more, subject to testing

Table 11. Suggested starting context budgets (my suggestions, test on your own work)

11. MOE VERSUS DENSE MODELS

CHARACTERISTIC	DENSE	SPARSE MOE
Which weights run	The same ones for every token	A selected few per token
Capacity and compute	Closely tied	Partly separated
Weight storage	Total parameters	Total parameters
Placement	Simple	More complex
Offloading	Whole layers	Expert-specific options
Speed	Excellent when fully on the GPU	Depends on hardware and placement
Predictability	More consistent	Varies with routing and placement
Answer quality	Depends on the model	Depends on the model

Table 12. Dense and sparse MoE models compared

Dense models make sense for fast conversation, full GPU residency, high throughput, simple tasks and simple deployments. MoE makes sense when more parameter capacity helps, a CPU and GPU split is acceptable, and some extra latency is tolerable. Research, software design, document analysis, multistep reasoning and unattended jobs are good candidates. Pick based on measured task quality, not architecture alone.

12. MOE AND THE MODEL RACK

In Local AI Model Racks I argued for a small set of complementary models instead of one ever-bigger general model [16]. It's easy to confuse the two kinds of routing.

LAYER	WHAT ROUTES	WHAT GETS PICKED
Application	An orchestrator or workflow	A task or tool
Model rack	A written rule	A whole model
Inside an MoE model	A learned router	Experts, token by token
Runtime	Placement settings	GPU, CPU or memory tier

Table 13. Routing happens at different layers

Rack routing picks a complete model for a task. MoE routing picks internal subnetworks for each token. They're complementary, not interchangeable. An MoE model can be one

slot in a rack, and on a 12 GB card it makes residency decisions matter even more, because it leaves little room for anything else.

13. BENCHMARKING AN MOE MODEL HONESTLY

METRIC	WHY IT MATTERS
Time to first token	How responsive it feels
Prompt processing rate	How fast it reads long input
Generation rate	How fast it writes
End-to-end time	What the user actually waits
Peak VRAM and RAM	Capacity planning
Power at the wall	Running cost
Task accuracy and failure rate	Whether the answers are any good

Table 14. What to measure

A fair comparison holds everything else equal: the same context, prompt lengths and generation lengths, one request at a time, comparable precision and the same runtime build, with warm-up runs and at least five repetitions. Test with thinking off and on separately. Record actual input and output token counts, because the same prompt can tokenize differently on different models. And don't infer quality from speed.

14. OPTIMIZING ON CONSTRAINED HARDWARE

Here's the order I'd work in. Change one thing at a time, and keep a baseline you can roll back to.

1. **Quantization.** Saves memory, at some risk to quality.
2. **Right-size the context.** Less cache, shorter history.
3. **GPU layer placement.** Faster, until you run out of VRAM.
4. **Expert-specific placement.** More flexibility, more CPU work.
5. **Optimized attention and cache quantization.** Less memory, some compatibility and precision risk.
6. **CPU thread tuning.** Better efficiency, until threads fight each other.
7. **Limit concurrency.** More predictable, less total throughput.

8. **Route work outside the model.** Send each job to the right model, at the cost of more moving parts.

GPU utilization is a diagnostic, not the goal. A GPU at 100 percent isn't necessarily doing useful work efficiently, and low utilization may point to the CPU, memory or synchronization. What counts is correct answers, finished tasks, acceptable latency and stability.

15. THE ECONOMICS OF MOE

Sparse computation doesn't automatically mean a lower power bill. A slower model can use more energy for the same output even if it draws fewer watts, because it runs longer. Energy in kilowatt-hours is watts x hours / 1,000. Cost is energy times your rate.

GENERATION SPEED	HOURS	ENERGY
5.57 tokens per second	About 49.9	About 8.0 kWh
11.75 tokens per second	About 23.6	About 3.8 kWh
19 tokens per second	About 14.6	About 2.3 kWh

Table 15. Illustration only: one million generated tokens at an assumed 160 W (calculated, not measured)

The wattage is an assumption, not a wall-meter reading, and the math covers generation only. It leaves out idle time, prompt processing, loading, cooling and the hardware itself. The point is that tuning placement cut the energy per token as much as it cut the wait.

The better yardstick is cost per successful task: total cost divided by tasks done right. A fast model that makes repeated mistakes can cost more than a slower one that gets it right. A large MoE with no measurable quality advantage can waste resources.

16. LIMITATIONS AND ENGINEERING CHALLENGES

- **Routing imbalance.** Experts can get uneven traffic in training and in use. Balancing strategies vary by model [1] [2].
- **Placement complexity.** Every memory tier adds transfer and synchronization costs, and the benefit of offloading varies by model [8].
- **Quantization sensitivity.** The router, experts, attention and shared parts may react differently to lower precision. Check quality after you quantize.

- **Runtime maturity.** Performance depends on the engine, its version, the backend and how it was built.
- **Concurrency.** Sessions can share weights, but each one adds cache and buffers. On a tight system that leads to instability. On one evening in November, my model process was stopped under memory pressure, which made in-flight requests error out.
- **Model intelligence.** Training data, tuning, distillation, quantization and the workload decide usefulness. A 35B MoE isn't equivalent to a dense 35B or a dense 3B.

17. SECURITY AND THE MODEL SUPPLY CHAIN

Community builds and quantized files add supply chain dependencies. Keep a manifest for every model you run.

FIELD	WHY
Publisher and repository	Who made this file
Revision	Which exact version you downloaded
Base model	What it was built from
Quantization	How it was compressed
Checksum	Proof the file hasn't changed
License	What you're allowed to do with it
Evaluation results	How it did on your own tests

Table 16. What a model manifest should record

A checksum proves a file's identity and integrity. It doesn't prove the model is free of poisoning or hidden triggers. Combine trusted publishers, documented lineage, isolated behavior tests, restricted tool permissions and ongoing monitoring, especially for agents that can use tools.

18. WHAT THIS MEANS FOR DECENTRALIZED AI

Sparse architectures can stretch the useful life of repurposed hardware and delay replacement costs. They don't remove the limits of old hardware, and they don't make a home lab competitive with a data center.

Large MoE models have been trained by spreading experts across many devices. Google's GShard spread a translation model of more than 600 billion parameters across 2,048 TPU accelerators [18]. Doing something similar across a few home machines brings its own problems in communication, placement, synchronization, scheduling, fault tolerance and security, and ordinary Ethernet isn't a dedicated accelerator interconnect. I haven't built anything like that. For private assistants, document processing, offline help, education and small-office work, though, a right-sized local system may beat trying to copy hyperscale capability.

19. FINDINGS AND CONCLUSION

1. **Capacity and computation are partly independent.** MoE separates total capacity from the parameters each token uses.
2. **Sparse activation doesn't eliminate storage.** A 35B-A3B model still needs room for all 35 billion parameters.
3. **Consumer hardware can run bigger models.** A file of about 22 GB runs on a 12 GB GPU with 32 GB of system memory.
4. **Sparse computation doesn't guarantee speed.** A dense model that fit on the GPU read long prompts far faster.
5. **Placement can matter as much as the model.** The same file ran at about 5.57 and 11.75 tokens per second in benchmarks on the same night.
6. **Context settings matter.** Native capacity isn't the same as what you allocate.
7. **Model racks and MoE work together.** They solve different allocation problems.
8. **Useful intelligence is the real metric.** Judge by correct answers, reliability and cost per successful task.

MoE changes how engineers think about model capacity and computation. Rather than throwing computation at everything, a learned router spends it selectively. That's what lets a large parameter pool work with a small active subset.

CyberTron shows the promise and the limits. An aging FX-8300, 32 GB of DDR3 and a 12 GB RTX 3060 run a 35B-total model every day. That doesn't beat enterprise hardware. It challenges the assumption about how much hardware you need to start. Physics still wins in the end. Memory bandwidth, GPU capacity, the PCIe link, the runtime and the context setting decide what you actually get.

The objective isn't to build the biggest AI system. It's to build the smartest infrastructure for running AI. Work smarter, not harder.

SOURCES

All links checked Nov. 24, 2026.

[1] Noam Shazeer and others, Google Brain, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," arXiv:1701.06538, 2017. https://arxiv.org/abs/1701.06538

[2] William Fedus, Barret Zoph and Noam Shazeer, "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity," arXiv:2101.03961, 2021. https://arxiv.org/abs/2101.03961

[3] Albert Q. Jiang and others, "Mixtral of Experts," arXiv:2401.04088, 2024. https://arxiv.org/abs/2401.04088

[4] Damai Dai and others, "DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models," arXiv:2401.06066, 2024. https://arxiv.org/abs/2401.06066

[5] Qwen, Qwen3.6-35B-A3B model page, Pirate Face. https://pirateface.co/Qwen/Qwen3.6-35B-A3B

[6] Qwen, Qwen3-30B-A3B model page, Pirate Face. https://pirateface.co/Qwen/Qwen3-30B-A3B

[7] llama.cpp, server README (GPU layers and MoE CPU placement options). https://github.com/ggml-org/llama.cpp/blob/master/tools/server/README.md

[8] Jingcong Liang and others, "Not All Models Suit Expert Offloading: On Local Routing Consistency of Mixture-of-Expert Models," arXiv:2505.16056, 2025. https://arxiv.org/abs/2505.16056

[9] TechSpot, NVIDIA GeForce RTX 3060 12 GB specifications (360 GB/s GDDR6 bandwidth). https://www.techspot.com/specs/gpu/223291-nvidia-geforce-rtx-3060-12gb.html

[10] DDR3 SDRAM reference article (DDR3-1600 is 12.8 GB/s per 64-bit channel). https://en.wikipedia.org/wiki/DDR3_SDRAM

[11] Bulldozer microarchitecture reference article (dual-channel DDR3 memory controller on desktop FX processors). https://en.wikipedia.org/wiki/Bulldozer_(microarchitecture)

[12] PCI Express link throughput by generation, PCI-SIG figures as summarized in the PCI Express reference article. https://en.wikipedia.org/wiki/PCI_Express

[13] Hugging Face, "Mixture of Experts Explained." https://huggingface.co/blog/moe

[14] Roggie D, "Bigger Isn't Always Smarter," Roger's Rabbit Hole, November 2026. https://www.rogersrabbithole.com/Rabbit-Hole/ArtMID/391/ArticleID/97/Bigger-Isnt-Always-Smarter

[15] Roggie D, "Building AI on Aging Hardware," Roger's Rabbit Hole, November 2026. https://www.rogersrabbithole.com/Rabbit-Hole/ArtMID/391/ArticleID/99/Building-AI-on-Aging-Hardware

[16] Roggie D, "Local AI Model Racks," Roger's Rabbit Hole, November 2026. https://www.rogersrabbithole.com/Rabbit-Hole/ArtMID/391/ArticleID/106/Local-AI-Model-Racks

[17] Roggie D, "Building CyberTron: A Post Said 50 Tokens a Second. My $285 GPU Did 5.56, Then 11.75," Roger's Rabbit Hole, November 2026. https://www.rogersrabbithole.com/Rabbit-Hole/ArtMID/391/ArticleID/105/Building-CyberTron-A-Post-Said-50-Tokens-a-Second-My-285-GPU-Did-556-Then-1175

[18] Dmitry Lepikhin and others, Google, "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding," arXiv:2006.16668, 2020. https://arxiv.org/abs/2006.16668

[19] List of AMD FX processors, reference article (four modules, two integer cores each, in the FX-8 series). https://en.wikipedia.org/wiki/List_of_AMD_FX_processors

Test note: The Oct. 16 figures are llama.cpp benchmark averages of three runs each, generating text from an empty prompt with no context size recorded, on an RTX 3060 12 GB, an AMD FX-8300 and 32 GB of DDR3. The Oct. 19 figures come from one real request on the early setup at an 8K context. The Oct. 21 figures come from one evaluation run on the same hardware at a 16K context, with three-run benchmark averages for speed and single-run suite scores. The November figures are single observations on a different build and settings. Calculated and illustrative figures are labeled where they appear.

ABOUT THE AUTHOR

Roggie D has been messing with computers since the early 1980s and working in IT since the mid-1990s, from network engineering to the CEO's chair. These days he's retired, which mostly means he gets to pick which problems are worth solving. He writes Roger's Rabbit Hole, where CyberTron, his budget local AI build, keeps sending him down new ones.

More from the Rabbit Hole at rogersrabbithole.com.