A Strategic Guide to Accelerating On-Device LLM Inference in the PocketLLM Application

Executive Summary: A Strategic Path to 10-20x Inference Speed on the S24+

The reported performance of approximately one token per second for local Large Language Model (LLM) inference within the PocketLLM application on a Samsung Galaxy S24+ device represents a critical performance bottleneck. This level of speed renders the application functionally unusable for interactive chat. Analysis indicates this is not a matter of minor optimization but a fundamental misconfiguration in how the application leverages the device's hardware. The root cause is the on-device inference engine operating in a generic, CPU-only mode, failing to utilize the powerful Graphics Processing Unit (GPU) and specialized hardware instructions available in the S24+'s System-on-Chip (SoC).

A three-pronged strategic framework is proposed to resolve this bottleneck and unlock the hardware's full potential. First, the underlying native inference engine, identified as a llama.cpp derivative, must be recompiled from source with build flags that explicitly enable hardware acceleration for the device's specific SoC. Second, the current LLM should be replaced with a smaller, highly efficient model (e.g., 1 to 4 billion parameters) that has been aggressively quantized to a 4-bit format, such as Q4_K_M, to reduce memory footprint and bandwidth requirements. Third, this newly compiled, hardware-aware library must be integrated back into the PocketLLM Android project, replacing the generic version, and tuned to offload the maximum number of model layers to the GPU.

Based on community benchmarks and the technical specifications of the S24+ hardware, implementing this strategy is projected to increase inference speed from the current ~1 token/second to a target range of **15 to 30+ tokens/second**. This represents a greater than 15-fold performance improvement, transforming the application's user experience from frustratingly slow to highly interactive and responsive.

Deconstructing the Performance Bottleneck: An Architectural Review

To effectively address the performance deficit, a thorough analysis of both the application's software stack and the target device's hardware architecture is necessary. The current slow speed is a direct result of a mismatch between these two components.

Analysis of the PocketLLM Application Stack

The PocketLLM application is architected as a cross-platform solution, pairing a Flutter-based user interface with a Python FastAPI backend for managing remote model providers like

OpenAI and Groq. However, for the use case of running a local LLM directly on the mobile device, the FastAPI and Supabase components are not part of the execution path. The performance bottleneck lies entirely within the on-device inference pipeline.
An examination of the project's source code composition reveals a native C++ component, accounting for 2.2% of the codebase, managed by CMake build scripts (1.7%). This is the native core responsible for the computationally intensive task of LLM inference. The Flutter front-end, written in Dart, communicates with this C++ library through a Foreign Function Interface (FFI), a standard methodology for integrating native code into Flutter applications. Given the project's focus on local LLMs and its associated tags (local-llm, llm-inference), the native engine is almost certainly an implementation of the popular llama.cpp library. This C++ library is the de facto standard for running GGUF-formatted models efficiently on consumer hardware. The critical issue arises from the "compatibility versus performance" trade-off inherent in distributing applications. To ensure an application runs on the widest possible range of Android devices, developers typically compile native libraries against a generic ARMv8-A instruction set, without enabling hardware-specific features. This generic binary fails to leverage the advanced capabilities of a high-end SoC like the one in the S24+, resulting in a fallback to a slow, unoptimized CPU execution path.

The Samsung S24+ Hardware Dichotomy: Snapdragon vs. Exynos

A crucial factor complicating any optimization effort is that the Samsung Galaxy S24+ is not a single hardware platform. It ships with two distinct SoC variants depending on the region of sale. These variants feature fundamentally different GPU architectures, mandating two separate optimization strategies. The first and most critical step for any developer or user is to identify which SoC their specific device contains.

Path A - Qualcomm Snapdragon 8 Gen 3

Found in North America and parts of East Asia, this variant is built around Qualcomm's flagship SoC.

- **CPU:** The octa-core processor features a high-performance ARM Cortex-X4 core, five Cortex-A720 cores, and two Cortex-A520 cores. It supports advanced instruction sets like ARM NEON and i8mm dot product, which can significantly accelerate certain quantized matrix operations even on the CPU.
- **GPU:** The Adreno 750 GPU is the primary target for performance acceleration. The key to unlocking its power for LLM inference is the OpenCL (Open Computing Language) framework. Qualcomm has recently contributed a highly optimized OpenCL backend directly to the llama.cpp project, specifically designed for Adreno GPUs. This makes OpenCL the premier path for Snapdragon-based devices.
- **NPU:** The SoC also includes a Hexagon Neural Processing Unit (NPU) designed for efficient AI/ML workloads.

Path B - Samsung Exynos 2400

This variant is used in most international markets.

- **CPU:** The deca-core processor uses a similar ARM architecture, with one Cortex-X4 core, five Cortex-A720 cores (split into two frequency clusters), and four Cortex-A520 cores.
- **GPU:** The Samsung Xclipse 940 GPU is fundamentally different from the Adreno. It is

based on AMD's RDNA graphics architecture, the same family used in desktop gaming GPUs. This architectural difference means that the Adreno-optimized OpenCL backend is not suitable. For RDNA-based GPUs on Android, the Vulkan graphics and compute API is the more viable and performant backend within the llama.cpp ecosystem.
- **NPU:** This SoC includes Samsung's proprietary NPU.

The existence of these two distinct hardware profiles means there can be no universal optimization solution for the S24+. An OpenCL-based library compiled for the Adreno GPU will be suboptimal or non-functional on the Xclipse GPU, and a Vulkan-based library will not leverage the specific optimizations available for Adreno. Therefore, any effective performance enhancement strategy must begin with hardware identification and proceed down one of two parallel implementation paths.

The Foundation of High-Speed Inference: Model Selection and Quantization

Before addressing hardware acceleration, optimizing the model itself is a prerequisite for achieving high performance on a resource-constrained device. The choice of model size and its quantization level directly impacts memory usage, bandwidth requirements, and computational speed.

Understanding the GGUF Format and Quantization

GGUF (Georgi Gerganov Universal Format) is the standard file format used by the llama.cpp ecosystem. It is a self-contained format that packages the model's architecture, its trained weights, and the necessary tokenizer information into a single file. This portability makes it ideal for on-device inference, as it simplifies model distribution and loading.
Quantization is a model compression technique that reduces the numerical precision of the model's weights. Instead of storing each weight as a 16-bit floating-point number (FP16), they are converted to lower-precision integers, such as 4-bit (INT4) or 8-bit (INT8). This process yields two primary benefits:

1. **Reduced Memory Footprint:** A 4-bit quantized model is roughly one-quarter the size of its FP16 counterpart, drastically reducing both storage requirements and the amount of RAM needed during inference.
2. **Faster Computation:** Integer arithmetic is significantly faster than floating-point arithmetic, especially on modern CPUs and GPUs that have specialized hardware instructions for handling low-precision data.

A Practical Guide to Quantization Methods

The llama.cpp project supports a wide array of quantization methods, often identified by a specific naming scheme. Understanding this scheme is key to selecting the right model file. For a typical format like Q4_K_M:

- Q4: Denotes a 4-bit quantization level.
- _K_: Refers to the "k-quants" method, an advanced technique that uses larger block sizes and more sophisticated scaling to preserve model quality more effectively than older methods.
- _M: Stands for "Medium," indicating the super-block size used, offering a balanced

trade-off within the k-quant framework.

The choice of quantization level involves a direct trade-off between performance and model quality (perplexity). Community benchmarks and empirical testing provide a clear hierarchy :

- **Q8_0:** An 8-bit quant that is considered nearly lossless in quality compared to the original FP16 model. It offers some speedup but remains large.
- **Q6_K / Q5_K_M:** 6-bit and 5-bit k-quants that represent a sweet spot for desktop hardware, offering significant size reduction with almost no perceptible degradation in quality.
- **Q4_K_M:** A 4-bit k-quant that is widely considered the **optimal balance for mobile devices**. It provides a dramatic reduction in size and a major boost in speed, while the quality loss is generally acceptable for most chat and instruction-following tasks.
- **Q3_K / Q2_K:** 3-bit and 2-bit quants that should be avoided unless memory constraints are extreme. They introduce significant quality degradation, leading to less coherent, shorter, and more repetitive responses.

Recommended Models for the Samsung S24+

The relationship between model size and hardware performance is symbiotic. Mobile SoCs are fundamentally limited by their shared memory bandwidth. LLM inference is a memory-bandwidth-bound task; the speed is often dictated by how quickly the model's weights can be fed to the compute units. A smaller model, even if slightly less "intelligent" in absolute terms, will always run faster because it requires less data to be transferred from RAM for each generated token. For this reason, selecting the smallest model that meets the required quality threshold is paramount for achieving a responsive experience.

The following models are recommended for deployment on the Samsung S24+, using the Q4_K_M quantization where available to maximize performance.

- **Top Recommendation: Microsoft Phi-3-mini (3.8B)** This model has demonstrated state-of-the-art performance among models of its size, offering strong reasoning and language capabilities in a compact package. It is the ideal starting point for a high-quality, responsive on-device assistant. The Phi-3-mini-4k-instruct-GGUF version is recommended.
- **High-Speed Alternative: Google Gemma 2B** For use cases where maximum speed is prioritized over complex reasoning, the 2-billion parameter Gemma model is an excellent choice. Its smaller size will yield a higher token-per-second rate compared to Phi-3-mini.
- **Ultra-Lightweight Option: TinyLlama 1.1B** When RAM is extremely limited or the highest possible inference speed is required for simple tasks, the TinyLlama model is a viable option. However, there is a noticeable drop in its reasoning and instruction-following capabilities. For this model, a higher-quality quantization like Q6_K or Q8_0 may be preferable to preserve as much of its limited capability as possible.

Model Name	Parameter Size	Recommended Quantization	File Size (GB)	Est. RAM Usage (GB)	Expected Performance (t/s)	Quality / Use-Case Notes
Phi-3-mini-Instruct	3.8B	Q4_K_M	~2.32	~2.8	15 - 25	**Top Recommendation.** Excellent balance of

Model Name	Parameter Size	Recommended Quantization	File Size (GB)	Est. RAM Usage (GB)	Expected Performance (t/s)	Quality / Use-Case Notes
						quality and performance for general chat and instruction-following.
Gemma-Instruct	2B	Q4_K_M	~1.5	~1.8	20 - 30+	**High-Speed Choice.** Faster than Phi-3 but with slightly lower reasoning capability. Ideal for simpler tasks.
Llama-3-Instruct	8B	Q4_K_M	~4.7	~5.7	8 - 15	Higher quality but may be slower due to memory bandwidth limits. A good option if Phi-3 is insufficient.
TinyLlama-Chat	1.1B	Q6_K	~0.8	~1.0	30 - 50+	**Ultra-Lightweight.** Very fast but significantly less capable. Suitable for basic tasks or memory-starved environments.

Unleashing Hardware Acceleration: A Practical Implementation Guide

The most substantial performance improvements are not achieved through runtime settings but

are unlocked during the compilation of the native llama.cpp library. The generic, pre-compiled library included with the PocketLLM application is almost certainly not built with the necessary flags to enable hardware acceleration on the S24+. Therefore, replacing this library with a custom-compiled version is the central task in resolving the performance bottleneck.

The "Compilation is the Optimization" Principle

The llama.cpp build system uses flags to conditionally compile support for various hardware backends, such as NVIDIA's CUDA, Apple's Metal, OpenCL, and Vulkan. These backends are disabled by default to create a universally compatible binary that depends only on the CPU. An app distributed through an official store will use such a generic build to ensure it functions on all devices. This explains the observed ~1 token/second performance, which is characteristic of CPU-only inference. To enable GPU acceleration, one cannot simply change a setting in the app; one must recompile the native library from source with the correct flags for the target hardware.

Environment Setup for Cross-Compilation

To build the Android library, a development environment with the following tools is required:

- **Android Studio:** For the Android Native Development Kit (NDK), which contains the toolchains for cross-compiling C++ code for Android.
- **CMake and Ninja:** The build system and generator used by llama.cpp.
- **Git:** For cloning the llama.cpp source code repository.

The path to the NDK must be available as an environment variable (e.g., ANDROID_NDK_HOME) for the build scripts to locate the toolchain.

Path A: Accelerating the Snapdragon 8 Gen 3 (Adreno GPU) via OpenCL

This is the recommended path for Snapdragon-based devices, as it leverages the purpose-built, highly optimized OpenCL backend provided by Qualcomm for its Adreno GPUs.

1. **Clone the llama.cpp repository:** git clone https://github.com/ggml-org/llama.cpp.git cd llama.cpp
2. **Create a build directory:** mkdir build-android-snapdragon cd build-android-snapdragon
3. **Configure the build with CMake:** Execute the following command, replacing $ANDROID_NDK_HOME with the path to the NDK. Each flag is critical for a successful, optimized build.

```
cmake.. \

-DCMAKE_TOOLCHAIN_FILE="$ANDROID_NDK_HOME/build/cmake/android.tool
chain.cmake" \
    -DANDROID_ABI=arm64-v8a \
    -DANDROID_PLATFORM=android-29 \
    -DGGML_OPENCL=ON \
    -DGGML_OPENCL_USE_ADRENO_KERNELS=ON \
    -DCMAKE_BUILD_TYPE=Release
```

    - -DGGML_OPENCL=ON: Enables the OpenCL backend.
    - -DGGML_OPENCL_USE_ADRENO_KERNELS=ON: **This is the most important flag.** It instructs the build system to compile the specialized kernels optimized for the Adreno GPU architecture.
4. **Compile the library:** cmake --build. --config Release -j$(nproc)
5. **Verify the output:** The compiled shared library, libllama.so, will be located in the build-android-snapdragon/bin/ directory.

Path B: Accelerating the Exynos 2400 (Xclipse GPU) via Vulkan

For Exynos devices, the Xclipse 940 GPU's AMD RDNA architecture makes the Vulkan backend the appropriate choice for acceleration.

1. **Follow the same initial steps as Path A** (clone repository, create a separate build directory like build-android-exynos).
2. **Configure the build with CMake using Vulkan flags:**

```
cmake.. \

-DCMAKE_TOOLCHAIN_FILE="$ANDROID_NDK_HOME/build/cmake/android.toolchain.cmake" \
    -DANDROID_ABI=arm64-v8a \
    -DANDROID_PLATFORM=android-29 \
    -DGGML_VULKAN=ON \
    -DCMAKE_BUILD_TYPE=Release
```

    - -DGGML_VULKAN=ON: Enables the Vulkan compute backend.
3. **Compile the library and verify the output** as in Path A. The resulting libllama.so will be specifically built with Vulkan support.

The Untapped Potential of the NPU: A Forward-Looking Analysis

While both the Snapdragon and Exynos SoCs contain powerful NPUs, leveraging them for llama.cpp is not a straightforward process. Unlike the GPU backends, there is no simple build flag to enable NPU acceleration. NPU integration typically requires vendor-specific SDKs and frameworks, such as Qualcomm's AI Engine Direct via GENIE or Google's MediaPipe LLM Inference API. These frameworks often demand specific model conversion steps and represent a significant architectural deviation from the existing llama.cpp-based structure of PocketLLM. While NPUs offer immense potential for power-efficient inference, the GPU acceleration path (via OpenCL for Snapdragon or Vulkan for Exynos) is directly supported by the llama.cpp build system and represents the most direct and achievable route to massive performance gains for this project.

Integrating the Optimized Engine into PocketLLM

After successfully compiling a hardware-accelerated version of the libllama.so library, the final step is to integrate it into the PocketLLM application, replacing the existing generic library.

Locating the Integration Point

Navigate the PocketLLM source code to the Android project directory, typically located at android/. Within this directory, there will be a CMakeLists.txt file that defines how the native C++ component is built and linked. This file is the primary integration point.

Replacing the Inference Library

The modification process involves instructing the build system to use the pre-compiled library instead of building one from source.

1. **Modify CMakeLists.txt:** Comment out or remove the existing add_subdirectory or source file list for the current llama.cpp implementation. Add the following lines to import the newly compiled shared library:
   ```
   # Add the pre-compiled shared library
   add_library(llama SHARED IMPORTED)
   set_target_properties(llama PROPERTIES IMPORTED_LOCATION

   ${CMAKE_SOURCE_DIR}/app/src/main/jniLibs/${ANDROID_ABI}/libllama.so)
   ```

2. **Place the Library:** Copy the libllama.so file compiled in the previous section into the android/app/src/main/jniLibs/arm64-v8a/ directory within the PocketLLM project.
3. **Update build.gradle:** Ensure the android/app/build.gradle file is configured to package the native libraries from the jniLibs directory.

Enabling GPU Offloading in Code

With the accelerated library in place, the application must be instructed to use it. This is done by setting the n_gpu_layers parameter, which tells llama.cpp how many layers of the neural network to offload to the GPU for processing.

1. **Locate Parameter Initialization:** Find the C++ source file within the project that initializes the llama.cpp model. This code will set up a llama_model_params struct.
2. **Set n_gpu_layers:** Within this initialization code, set the n_gpu_layers field. A high value, such as 99, is a common practice to instruct the engine to offload as many layers as possible.
   ```
   struct llama_model_params model_params =
   llama_model_default_params();
   //... other params
   model_params.n_gpu_layers = 99; // Offload all possible layers to
   GPU
   ```

3. **Expose as a Setting (Recommended):** For optimal flexibility, it is highly recommended to modify the FFI bridge to allow this n_gpu_layers value to be set from the Flutter UI. This creates a crucial tuning knob for the end-user to balance performance against memory usage and battery consumption without requiring a recompile of the application.

Benchmarking, Tuning, and Final Recommendations

With the optimized library integrated, a systematic process of benchmarking and tuning is required to confirm performance gains and find the optimal configuration for the device.

Measuring Performance with llama-bench

Before integrating the library into the full Flutter application, its raw performance should be verified using the llama-bench executable, which is compiled alongside libllama.so. This command-line tool provides precise metrics without the overhead of the application UI.

1. **Push Files to Device:** Use the Android Debug Bridge (adb) to push llama-bench, the model file (e.g., phi-3-mini-q4_k_m.gguf), and the compiled libllama.so to a directory on the device.
2. **Run Benchmark:** Execute the benchmark from an adb shell session : ./llama-bench -m phi-3-mini-q4_k_m.gguf -ngl 35 -t 4
    - -m: Specifies the model file.
    - -ngl: Sets the number of GPU layers to offload.
    - -t: Sets the number of CPU threads to use.
3. **Analyze Output:** The output will contain critical information. For Snapdragon devices, a confirmation message like ggml_opencl: using kernels optimized for Adreno verifies that the correct backend is active. Key performance metrics include:
    - **Time to First Token (TTFT):** Measures prompt processing speed and is crucial for the perception of responsiveness.
    - **Tokens per Second (t/s):** Measures the sustained generation speed, the primary metric for chat performance.

A Practical Tuning Workflow

The optimal number of GPU layers (n_gpu_layers) depends on the model size and the device's available VRAM and memory bandwidth. A systematic approach is needed to find the peak performance point.

1. **Establish Baselines:** Run llama-bench with -ngl 0 to measure the pure CPU performance.
2. **Increment Offloading:** Rerun the benchmark, increasing the -ngl value in increments (e.g., 5, 10, 15, 20...).
3. **Record Performance:** At each step, record the resulting tokens per second.
4. **Identify the Peak:** Performance will typically increase with more layers on the GPU until it plateaus or even slightly decreases. This peak occurs when the overhead of managing memory transfer outweighs the computational benefit, or when the device begins to thermally throttle. The -ngl value at this peak is the optimal setting for that specific model on that device.

Final Tiered Recommendations

To provide a clear path forward, the following tiered recommendations are provided, ordered from lowest effort to highest impact.

- **Tier 1 (Immediate, Low Effort): Model Optimization.** Before modifying any code, the simplest first step is to switch to a more appropriate model. Download a Q4_K_M quantized version of a small, high-performance model like **Phi-3-mini (3.8B)**. Using this model, even with the existing CPU-only engine, will likely result in a 2-4x performance improvement due to the significantly reduced computational and memory load.
- **Tier 2 (Moderate Effort, High Impact): Full Hardware Acceleration.** This is the core solution. Follow the complete guide outlined in Sections IV and V to identify the device's SoC, cross-compile the llama.cpp library with the correct GPU backend (OpenCL for Snapdragon, Vulkan for Exynos), and integrate this optimized library into the PocketLLM application. This is the definitive path to achieving the target performance of **15-30+ tokens per second**.
- **Tier 3 (Advanced Tuning and Future-Proofing):** Once Tier 2 is complete, further refinements are possible. Experiment with different advanced quantization methods (such as the IQ series) which may offer better quality at small sizes. Periodically pull updates from the main llama.cpp repository and recompile, as performance improvements are frequently added. Monitor the open-source community for advancements in Vulkan performance on RDNA GPUs and for more accessible NPU integration paths in the future.

Works cited

1. How To Cmake Llama.cpp Build For Adreno 750 GPU Snapdragon 8 Gen 3? - Reddit, https://www.reddit.com/r/termux/comments/1mxrire/how_to_cmake_llamacpp_build_for_adreno_750_gpu/ 2. Best Local LLM app for mobile? : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1hr5hai/best_local_llm_app_for_mobile/ 3. Running LLM on Android (Snapdragon 8 Gen 3) : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1bpw9c7/running_llm_on_android_snapdragon_8_gen_3/ 4. PocketLLM/PocketLLM: A powerful Flutter-based AI chat ... - GitHub, https://github.com/Mr-Dark-debug/PocketLLM 5. local-llm · GitHub Topics, https://github.com/topics/local-llm?l=dart&o=desc&s=updated 6. ollama · GitHub Topics · GitHub, https://github.com/topics/ollama?l=dart&o=asc&s=forks 7. Samsung Galaxy S24+: Specs, Features & Everything you need to know - Pocketnow, https://pocketnow.com/samsung-galaxy-s24-plus/ 8. Samsung Galaxy S24 - Wikipedia, https://en.wikipedia.org/wiki/Samsung_Galaxy_S24 9. Samsung Galaxy S24+ - Full phone specifications - GSMArena.com, https://www.gsmarena.com/samsung_galaxy_s24+-12772.php 10. Samsung Galaxy S24+ Full Specifications - PhoneArena, https://www.phonearena.com/phones/Samsung-Galaxy-S24+_id12114 11. [llama.cpp] Android users now benefit from faster prompt processing with improved arm64 support. : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1ebnkds/llamacpp_android_users_now_benefit_from_faster/ 12. Introducing the new OpenCL™ GPU Backend in llama.cpp for Qualcomm Adreno GPUs, https://www.droidcon.com/2025/02/06/introducing-the-new-opencl-gpu-backend-in-llama-cpp-for-qualcomm-adreno-gpus/ 13. Introducing the New OpenCL GPU Backend in llama.cpp for Qualcomm Adreno GPUs, https://www.qualcomm.com/developer/blog/2024/11/introducing-new-opn-cl-gpu-backend-llama-cpp-for-qualcomm-adreno-gpu 14. Exynos 2400 | Mobile Processor | Samsung Semiconductor Global, https://semiconductor.samsung.com/processor/mobile-processor/exynos-2400/ 15. How

to utilize GPU on Android to accelerate inference? · Issue #8705 · ggml-org/llama.cpp, https://github.com/ggml-org/llama.cpp/issues/8705 16. llama.cpp: The Ultimate Guide to Efficient LLM Inference and Applications - PyImageSearch, https://pyimagesearch.com/2024/08/26/llama-cpp-the-ultimate-guide-to-efficient-llm-inference-and-applications/ 17. TheBloke/TinyLlama-1.1B-Chat-v1.0-GGUF - Hugging Face, https://huggingface.co/TheBloke/TinyLlama-1.1B-Chat-v1.0-GGUF 18. Ultimate Guide to Running Quantized LLMs on CPU with LLaMA.cpp - Medium, https://medium.com/red-buffer/ultimate-guide-to-running-quantized-llms-on-cpu-with-llama-cpp-1a26c34bb6dd 19. Gemma 2b Gguf Quantized · Models - Dataloop, https://dataloop.ai/library/model/rahuldshetty_gemma-2b-gguf-quantized/ 20. llama.cpp - Qwen, https://qwen.readthedocs.io/en/latest/quantization/llama.cpp.html 21. Even more quantization types? #5063 - ggml-org llama.cpp - GitHub, https://github.com/ggml-org/llama.cpp/discussions/5063 22. Extensive llama.cpp benchmark for quality degradation by quantization - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1jjwj88/extensive_llamacpp_benchmark_for_quality/ 23. how much Quantization decrease model's capability? : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1ja3vjf/how_much_quantization_decrease_models_capability/ 24. Phi 3 Medium 4k Instruct GGUF · Models - Dataloop, https://dataloop.ai/library/model/bartowski_phi-3-medium-4k-instruct-gguf/ 25. QuantFactory/Phi-3-mini-128k-instruct-GGUF - Hugging Face, https://huggingface.co/QuantFactory/Phi-3-mini-128k-instruct-GGUF 26. QuantFactory/Phi-3-mini-4k-instruct-GGUF - Hugging Face, https://huggingface.co/QuantFactory/Phi-3-mini-4k-instruct-GGUF 27. Phi-3 is a family of lightweight 3B (Mini) and 14B (Medium) state-of-the-art open models by Microsoft. - Ollama, https://ollama.com/library/phi3 28. Gemma 7b It Gguf Quantized · Models - Dataloop, https://dataloop.ai/library/model/rahuldshetty_gemma-7b-it-gguf-quantized/ 29. TinyLlama 1.1B Chat V1.0 GGUF · Models - Dataloop, https://dataloop.ai/library/model/thebloke_tinyllama-11b-chat-v10-gguf/ 30. ggml-org/llama.cpp: LLM inference in C/C++ - GitHub, https://github.com/ggml-org/llama.cpp 31. README.md · 5a51cc1bb4592f0d71f9af89cd08b11a066ba447 · Till-Ole Herbst / Llama.Cpp - GitLab, https://gitlab.informatik.uni-halle.de/ambcj/llama.cpp/-/blob/5a51cc1bb4592f0d71f9af89cd08b11a066ba447/README.md 32. Accelerate LLAMA & LangChain with Local GPU - Leonid Olevsky - Medium, https://lolevsky.medium.com/running-locally-llama-and-langchain-accelerated-by-gpu-a52a2fd72d79 33. Building Llama On-Device With Qualcomm Gen AI Inference Extensions (GENIE) | by PAD Editorial | ProAndroidDev, https://proandroiddev.com/building-llama-on-device-with-qualcomm-gen-ai-inference-extensions-genie-45f4f6f8961b 34. LLM Inference guide for Android | Google AI Edge, https://ai.google.dev/edge/mediapipe/solutions/genai/llm_inference/android 35. Phi 3 Mini 4k Instruct V0.3 GGUF · Models - Dataloop, https://dataloop.ai/library/model/bartowski_phi-3-mini-4k-instruct-v03-gguf/