HOWTO: Schnelle eigene llama.cpp-Integration in Java (ohne Ollama)

1. Ziel

Dieses How■To erklärt, wie du llama.cpp in einer Java■Anwendung so einbindest,

dass du annähernd oder vollständig die Leistung von Ollama erreichst.

2. Voraussetzungen

• Linux oder Windows

• GCC/Clang

• CMake

• Java 17+

• RTX 3060 oder CPU■only

3. llama.cpp optimal kompilieren

CPU■optimiert:

cmake -B build -S . -DCMAKE_BUILD_TYPE=Release -DLLAMA_BLAS=ON
-DLLAMA_BLAS_VENDOR=OpenBLAS

cmake --build build -j

GPU■optimiert (RTX 3060):

cmake -B build -S . -DCMAKE_BUILD_TYPE=Release -DLLAMA_CUBLAS=ON
-DLLAMA_BLAS=ON -DLLAMA_BLAS_VENDOR=OpenBLAS

cmake --build build -j

Wichtig:

• -O3 und -march=native werden automatisch gesetzt.

• CUDA-Version muss zu deinem NVIDIA■Treiber passen.

4. Java■Integration per JNI

Wichtig ist ein „grober“ JNI■Aufruf:

nativeGenerate(prompt, maxTokens, settings)

Regeln:

• NICHT pro Token JNI aufrufen → stark langsam.

• Modell und Kontext nur EINMAL laden.

• JVM und llama■Threads nicht gegeneinander kämpfen lassen.

5. Kontext warm halten

• Modell beim Start laden

• Kontext(e) als Singleton oder Pool führen

• Kein Reload pro Anfrage

6. Leistungsgewinne

Je nach Ausgangslage erzielt man:

• 30–100 % allein durch besseren Build

• 3×–10× durch GPU■Nutzung

• 20–50 % durch weniger JNI■Calls

• Insgesamt: Java■Integration kann so schnell wie Ollama werden.

7. Empfehlungen für RTX 3060

• n_threads = Anzahl physischer Kerne

• n_gpu_layers = 30–40 für 7B, 50+ für größere Modelle

• Quantisierung q4_K_M für beste Kombination aus Geschwindigkeit/Qualität

8. Fazit

Mit optimiertem llama.cpp■Build, GPU■Offload und sauberem JNI■Design erreichst du vergleichbare oder bessere Geschwindigkeit als Ollama – völlig unabhängig vom Ollama■Service.