









 gggerganov / llama.cpp



 **Code**  **Issues** **262**  **Pull requests** **311**  **Discussions**  **Actions**  **Projects** **9** 

llama.cpp / docs / build.md 

yeahdongcn musa : update doc (#9856)



943d20b · last month



396 lines (292 loc) · 22.7 KB

Preview

Code

Blame

Raw



Build llama.cpp locally

To get the Code:

```
git clone https://github.com/gggerganov/llama.cpp
cd llama.cpp
```



In order to build llama.cpp you have four different options.

- Using `make` :

- On Linux or MacOS:

```
make
```



- On Windows (x86/x64 only, arm64 requires cmake):

- Download the latest fortran version of [w64devkit](#).
- Extract `w64devkit` on your pc.
- Run `w64devkit.exe` .
- Use the `cd` command to reach the `llama.cpp` folder.
- From here you can run:

```
make
```



- Notes:

- For `q4_0_4_4` quantization type build, add the `GGML_NO_LLAMAFILE=1` flag. For example, use `make GGML_NO_LLAMAFILE=1` .
- For faster compilation, add the `-j` argument to run multiple jobs in parallel. For example, `make -j 8` will run 8 jobs in parallel.

- For faster repeated compilation, install [ccache](#).
- For debug builds, run `make LLAMA_DEBUG=1`
- Using CMake :

```
cmake -B build
cmake --build build --config Release
```

Notes:

- For `Q4_0_4_4` quantization type build, add the `-DGGML_LLAMAFILE=OFF` cmake option. For example, use `cmake -B build -DGGML_LLAMAFILE=OFF`.
- For faster compilation, add the `-j` argument to run multiple jobs in parallel. For example, `cmake --build build --config Release -j 8` will run 8 jobs in parallel.
- For faster repeated compilation, install [ccache](#).
- For debug builds, there are two cases:

- a. Single-config generators (e.g. default = `Unix Makefiles` ; note that they just ignore the `--config` flag):

```
cmake -B build -DCMAKE_BUILD_TYPE=Debug
cmake --build build
```

- b. Multi-config generators (`-G` param set to Visual Studio, XCode...):

```
cmake -B build -G "Xcode"
cmake --build build --config Debug
```

- Building for Windows (x86, x64 and arm64) with MSVC or clang as compilers:
 - Install Visual Studio 2022, e.g. via the [Community Edition](#). In the installer, select at least the following options (this also automatically installs the required additional tools like CMake,...):
 - Tab Workload: Desktop-development with C++
 - Tab Components (select quickly via search): C++-CMake Tools for Windows, Git for Windows, C++-Clang Compiler for Windows, MS-Build Support for LLVM-Toolset (clang)
 - Please remember to always use a Developer Command Prompt / PowerShell for VS2022 for git, build, test
 - For Windows on ARM (arm64, WoA) build with:

```
cmake --preset arm64-windows-llvm-release -D GGML_OPENMP=OFF
cmake --build build-arm64-windows-llvm-release
```

Note: Building for arm64 could also be done just with MSVC (with the `build-arm64-windows-MSVC` preset, or the standard CMake build instructions). But

MSVC does not support inline ARM assembly-code, used e.g. for the accelerated Q4_0_4_8 CPU kernels.

- Using `gmake` (FreeBSD):
 - i. Install and activate [DRM in FreeBSD](#)
 - ii. Add your user to **video** group
 - iii. Install compilation dependencies.

```
sudo pkg install gmake automake autoconf pkgconf llvm15 openblas  
  
gmake CC=/usr/local/bin/clang15 CXX=/usr/local/bin/clang++15 -j4
```



Metal Build

On MacOS, Metal is enabled by default. Using Metal makes the computation run on the GPU. To disable the Metal build at compile time use the `GGML_NO_METAL=1` flag or the `GGML_METAL=OFF` cmake option.

When built with Metal support, you can explicitly disable GPU inference with the `--n-gpu-layers 0` command-line argument.

BLAS Build

Building the program with BLAS support may lead to some performance improvements in prompt processing using batch sizes higher than 32 (the default is 512). Support with CPU-only BLAS implementations doesn't affect the normal generation performance. We may see generation performance improvements with GPU-involved BLAS implementations, e.g. cuBLAS, hipBLAS. There are currently several different BLAS implementations available for build and use:

Accelerate Framework:

This is only available on Mac PCs and it's enabled by default. You can just build using the normal instructions.

OpenBLAS:

This provides BLAS acceleration using only the CPU. Make sure to have OpenBLAS installed on your machine.

- Using `make` :
 - On Linux:

```
make GGML_OPENBLAS=1
```



- On Windows:

- a. Download the latest fortran version of [w64devkit](#).
- b. Download the latest version of [OpenBLAS for Windows](#).
- c. Extract `w64devkit` on your pc.
- d. From the OpenBLAS zip that you just downloaded copy `libopenblas.a` , located inside the `lib` folder, inside `w64devkit\x86_64-w64-mingw32\lib` .
- e. From the same OpenBLAS zip copy the content of the `include` folder inside `w64devkit\x86_64-w64-mingw32\include` .
- f. Run `w64devkit.exe` .
- g. Use the `cd` command to reach the `llama.cpp` folder.
- h. From here you can run:

```
make GGML_OPENBLAS=1
```



- Using `CMake` on Linux:

```
cmake -B build -DGGML_BLAS=ON -DGGML_BLAS_VENDOR=OpenBLAS  
cmake --build build --config Release
```



BLIS

Check [BLIS.md](#) for more information.

SYCL

SYCL is a higher-level programming model to improve programming productivity on various hardware accelerators.

llama.cpp based on SYCL is used to **support Intel GPU** (Data Center Max series, Flex series, Arc series, Built-in GPU and iGPU).

For detailed info, please refer to [llama.cpp for SYCL](#).

Intel oneMKL

Building through oneAPI compilers will make `avx_vnni` instruction set available for intel processors that do not support `avx512` and `avx512_vnni`. Please note that this build config **does not support Intel GPU**. For Intel GPU support, please refer to [llama.cpp for SYCL](#).

- Using manual oneAPI installation: By default, `GGML_BLAS_VENDOR` is set to `Generic`, so if you already sourced intel environment script and assign `-DGGML_BLAS=ON` in cmake, the mkl version of Blas will automatically been selected. Otherwise please install oneAPI and follow the below steps:

```
source /opt/intel/oneapi/setvars.sh # You can skip this step if in oneapi-bas
cmake -B build -DGGML_BLAS=ON -DGGML_BLAS_VENDOR=Intel10_64lp -DCMAKE_C_COMPILER
cmake --build build --config Release
```

- Using oneAPI docker image: If you do not want to source the environment vars and install oneAPI manually, you can also build the code using intel docker container: [oneAPI-basekit](#). Then, you can use the commands given above.

Check [Optimizing and Running LLaMA2 on Intel® CPU](#) for more information.

CUDA

This provides GPU acceleration using the CUDA cores of your Nvidia GPU. Make sure to have the CUDA toolkit installed. You can download it from your Linux distro's package manager (e.g. `apt install nvidia-cuda-toolkit`) or from here: [CUDA Toolkit](#).

For Jetson user, if you have Jetson Orin, you can try this: [Offical Support](#). If you are using an old model(nano/TX2), need some additional operations before compiling.

- Using `make` :

```
make GGML_CUDA=1
```

- Using `CMake` :

```
cmake -B build -DGGML_CUDA=ON
cmake --build build --config Release
```

The environment variable `CUDA_VISIBLE_DEVICES` can be used to specify which GPU(s) will be used.

The environment variable `GGML_CUDA_ENABLE_UNIFIED_MEMORY=1` can be used to enable unified memory in Linux. This allows swapping to system RAM instead of crashing when the GPU VRAM is exhausted. In Windows this setting is available in the NVIDIA control panel as `System Memory Fallback`.

The following compilation options are also available to tweak performance:

Option	Legal values	Default	Description
GGML_CUDA_FORCE_DMMV	Boolean	false	Force the use of dequantization + matrix vector multiplication kernels instead of using kernels that do matrix vector multiplication on quantized data. By default the decision is made based on compute capability (MMVQ for 6.1/Pascal/GTX 1000 or higher). Does not affect k-quants.
GGML_CUDA_DMMV_X	Positive integer >= 32	32	Number of values in x direction processed by the CUDA dequantization + matrix vector multiplication kernel per iteration. Increasing this value can improve performance on fast GPUs. Power of 2 heavily recommended. Does not affect k-quants.
GGML_CUDA_MMV_Y	Positive integer	1	Block size in y direction for the CUDA mul mat vec kernels. Increasing this value can improve performance on fast GPUs. Power of 2 recommended.
GGML_CUDA_FORCE_MMQ	Boolean	false	Force the use of custom matrix multiplication kernels for quantized models instead of FP16 cuBLAS even if there is no int8 tensor core implementation available (affects V100, RDNA3). MMQ kernels are enabled by default on GPUs with int8 tensor core support.

Option	Legal values	Default	Description
			With MMQ force enabled, speed for large batch sizes will be worse but VRAM consumption will be lower.
GGML_CUDA_FORCE_CUBLAS	Boolean	false	Force the use of FP16 cuBLAS instead of custom matrix multiplication kernels for quantized models
GGML_CUDA_F16	Boolean	false	If enabled, use half-precision floating point arithmetic for the CUDA dequantization + mul mat vec kernels and for the q4_1 and q5_1 matrix multiplication kernels. Can improve performance on relatively recent GPUs.
GGML_CUDA_KQUANTS_ITER	1 or 2	2	Number of values processed per iteration and per CUDA thread for Q2_K and Q6_K quantization formats. Setting this value to 1 can improve performance for slow GPUs.
GGML_CUDA_PEER_MAX_BATCH_SIZE	Positive integer	128	Maximum batch size for which to enable peer access between multiple GPUs. Peer access requires either Linux or NVLink. When using NVLink enabling peer access for larger batch sizes is potentially beneficial.
GGML_CUDA_FA_ALL_QUANTS	Boolean	false	Compile support for all KV cache quantization type (combinations) for the FlashAttention CUDA

Option	Legal values	Default	Description
			kernels. More fine-grained control over KV cache size but compilation takes much longer.

MUSA

This provides GPU acceleration using the MUSA cores of your Moore Threads MTT GPU. Make sure to have the MUSA SDK installed. You can download it from here: [MUSA SDK](#).

- Using `make` :

```
make GGML_MUSA=1
```



- Using `CMake` :

```
cmake -B build -DGGML_MUSA=ON  
cmake --build build --config Release
```



The environment variable `MUSA_VISIBLE_DEVICES` can be used to specify which GPU(s) will be used.

The environment variable `GGML_CUDA_ENABLE_UNIFIED_MEMORY=1` can be used to enable unified memory in Linux. This allows swapping to system RAM instead of crashing when the GPU VRAM is exhausted.

Most of the compilation options available for CUDA should also be available for MUSA, though they haven't been thoroughly tested yet.

hipBLAS

This provides BLAS acceleration on HIP-supported AMD GPUs. Make sure to have ROCm installed. You can download it from your Linux distro's package manager or from here: [ROCm Quick Start \(Linux\)](#).

- Using `make` :

```
make GGML_HIPBLAS=1
```



- Using `CMake` for Linux (assuming a gfx1030-compatible AMD GPU):

```
HIPCCX="$(hipconfig -l)/clang" HIP_PATH="$(hipconfig -R)" \  
cmake -S . -B build -DGGML_HIPBLAS=ON -DAMDGPU_TARGETS=gfx1030 -DCMAKE_BUI
```




```
&& cmake --build build --config Release -- -j 16
```

On Linux it is also possible to use unified memory architecture (UMA) to share main memory between the CPU and integrated GPU by setting `-DGGML_HIP_UMA=ON`. However, this hurts performance for non-integrated GPUs (but enables working with integrated GPUs).

Note that if you get the following error:

```
clang: error: cannot find ROCm device library; provide its path via '--rocm-path' or '--rocm-device-lib-path', or pass '-nogpulib' to build without ROCm device library
```

Try searching for a directory under `HIP_PATH` that contains the file `oclc_abi_version_400.bc`. Then, add the following to the start of the command: `HIP_DEVICE_LIB_PATH=<directory-you-just-found>`, so something like:

```
HIPCCX="$(hipconfig -l)/clang" HIP_PATH="$(hipconfig -p)" \
HIP_DEVICE_LIB_PATH=<directory-you-just-found> \
  cmake -S . -B build -DGGML_HIPBLAS=ON -DAMDGPU_TARGETS=gfx1030 -DCMAKE_BUI
&& cmake --build build -- -j 16
```

- Using `make` (example for target gfx1030, build with 16 CPU threads):

```
make -j16 GGML_HIPBLAS=1 GGML_HIP_UMA=1 AMDGPU_TARGETS=gfx1030
```

- Using `CMake` for Windows (using x64 Native Tools Command Prompt for VS, and assuming a gfx1100-compatible AMD GPU):

```
set PATH=%HIP_PATH%\bin;%PATH%
cmake -S . -B build -G Ninja -DAMDGPU_TARGETS=gfx1100 -DGGML_HIPBLAS=ON -DCMAK
cmake --build build
```

Make sure that `AMDGPU_TARGETS` is set to the GPU arch you want to compile for. The above example uses `gfx1100` that corresponds to Radeon RX 7900XTX/XT/GRE. You can find a list of targets [here](#). Find your gpu version string by matching the most significant version information from `rocminfo | grep gfx | head -1 | awk '{print $2}'` with the list of processors, e.g. `gfx1035` maps to `gfx1030`.

The environment variable [HIP_VISIBLE_DEVICES](#) can be used to specify which GPU(s) will be used. If your GPU is not officially supported you can use the environment variable `[HSA_OVERRIDE_GFX_VERSION]` set to a similar GPU, for example 10.3.0 on RDNA2 (e.g. gfx1030, gfx1031, or gfx1035) or 11.0.0 on RDNA3. The following compilation options are also available to tweak performance (yes, they refer to CUDA, not HIP, because it uses the same code as the cuBLAS version above):

Option	Legal values	Default	Description
GGML_CUDA_DMMV_X	Positive integer ≥ 32	32	Number of values in x direction processed by the HIP dequantization + matrix vector multiplication kernel per iteration. Increasing this value can improve performance on fast GPUs. Power of 2 heavily recommended. Does not affect k-quants.
GGML_CUDA_MMV_Y	Positive integer	1	Block size in y direction for the HIP mul mat vec kernels. Increasing this value can improve performance on fast GPUs. Power of 2 recommended. Does not affect k-quants.
GGML_CUDA_KQUANTS_ITER	1 or 2	2	Number of values processed per iteration and per HIP thread for Q2_K and Q6_K quantization formats. Setting this value to 1 can improve performance for slow GPUs.

Vulkan

Windows

w64devkit

Download and extract [w64devkit](#).

Download and install the [Vulkan SDK](#). When selecting components, only the Vulkan SDK Core is required.

Launch `w64devkit.exe` and run the following commands to copy Vulkan dependencies:

```
SDK_VERSION=1.3.283.0
cp /VulkanSDK/$SDK_VERSION/Bin/glslc.exe $W64DEVKIT_HOME/bin/
cp /VulkanSDK/$SDK_VERSION/Lib/vulkan-1.lib $W64DEVKIT_HOME/x86_64-w64-mingw32/lib/
```



```
cp -r /VulkanSDK/$SDK_VERSION/Include/* $W64DEVKIT_HOME/x86_64-w64-mingw32/include,
cat > $W64DEVKIT_HOME/x86_64-w64-mingw32/lib/pkgconfig/vulkan.pc <<EOF
Name: Vulkan-Loader
Description: Vulkan Loader
Version: $SDK_VERSION
Libs: -lvulkan-1
EOF
```

Switch into the `llama.cpp` directory and run `make GGML_VULKAN=1`.

MSYS2

Install [MSYS2](#) and then run the following commands in a UCRT terminal to install dependencies.

```
pacman -S git \
    mingw-w64-ucrt-x86_64-gcc \
    mingw-w64-ucrt-x86_64-cmake \
    mingw-w64-ucrt-x86_64-vulkan-devel \
    mingw-w64-ucrt-x86_64-shaderc
```

Switch into `llama.cpp` directory and build using CMake.

```
cmake -B build -DGGML_VULKAN=ON
cmake --build build --config Release
```

With docker:

You don't need to install Vulkan SDK. It will be installed inside the container.

```
# Build the image
docker build -t llama-cpp-vulkan -f .devops/llama-cli-vulkan.Dockerfile .

# Then, use it:
docker run -it --rm -v "$(pwd):/app:Z" --device /dev/dri/renderD128:/dev/dri/rende
```

Without docker:

Firstly, you need to make sure you have installed [Vulkan SDK](#)

For example, on Ubuntu 22.04 (jammy), use the command below:

```
wget -qO - https://packages.lunarg.com/lunarg-signing-key-pub.asc | apt-key add -
wget -qO /etc/apt/sources.list.d/lunarg-vulkan-jammy.list https://packages.lunarg.com
apt update -y
apt-get install -y vulkan-sdk
```

```
# To verify the installation, use the command below:
vulkaninfo
```

Alternatively your package manager might be able to provide the appropriate libraries. For example for Ubuntu 22.04 you can install `libvulkan-dev` instead. For Fedora 40, you can install `vulkan-devel`, `glslc` and `glslang` packages.

Then, build llama.cpp using the cmake command below:

```
cmake -B build -DGGML_VULKAN=1
cmake --build build --config Release
# Test the output binary (with "-ngl 33" to offload all layers to GPU)
./bin/llama-cli -m "PATH_TO_MODEL" -p "Hi you how are you" -n 50 -e -ngl 33 -t 4

# You should see in the output, ggml_vulkan detected your GPU. For example:
# ggml_vulkan: Using Intel(R) Graphics (ADL GT2) | uma: 1 | fp16: 1 | warp size: 3:
```

CANN

This provides NPU acceleration using the AI cores of your Ascend NPU. And [CANN](#) is a hierarchical APIs to help you to quickly build AI applications and service based on Ascend NPU.

For more information about Ascend NPU in [Ascend Community](#).

Make sure to have the CANN toolkit installed. You can download it from here: [CANN Toolkit](#)

Go to `llama.cpp` directory and build using CMake.

```
cmake -B build -DGGML_CANN=on -DCMAKE_BUILD_TYPE=release
cmake --build build --config release
```

You can test with:

```
./build/llama-cli -m PATH_TO_MODEL -p "Building a website can be done in 10 steps:" -ngl 32
```

If the following info is output on screen, you are using `llama.cpp` by CANN backend :

```
llm_load_tensors:      CANN buffer size = 13313.00 MiB
llama_new_context_with_model:  CANN compute buffer size = 1260.81 MiB
```

For detailed info, such as model/device supports, CANN install, please refer to [llama.cpp for CANN](#).

Android

To read documentation for how to build on Android, [click here](#)

Arm CPU optimized mulmat kernels

Llama.cpp includes a set of optimized mulmat kernels for the Arm architecture, leveraging Arm® Neon™, int8mm and SVE instructions. These kernels are enabled at build time through the appropriate compiler cpu-type flags, such as `-DCMAKE_C_FLAGS=-march=armv8.2a+int8mm+sve`. Note that these optimized kernels require the model to be quantized into one of the formats: `Q4_0_4_4` (Arm Neon), `Q4_0_4_8` (int8mm) or `Q4_0_8_8` (SVE). The SVE mulmat kernel specifically requires a vector width of 256 bits. When running on devices with a different vector width, it is recommended to use the `Q4_0_4_8` (int8mm) or `Q4_0_4_4` (Arm Neon) formats for better performance. Refer to [examples/quantize/README.md](#) for more information on the quantization formats.

To support `Q4_0_4_4`, you must build with `GGML_NO_LLAMAFILE=1` (`make`) or `DGGML_LLAMAFILE=OFF` (`cmake`).