密级状态：绝密(　　)　　秘密(　　)　　内部(　　)　　公开(√)

RKLLM SDK User Guide

（技术部，图形计算平台中心）

文件状态： [　] 正在修改 [√] 正式发布	当前版本：	1.0.0
	作　　者：	AI 组
	完成日期：	2024-3-23
	审　　核：	熊伟
	完成日期：	2024-3-23

瑞芯微电子股份有限公司

Rockchips　Semiconductor　Co . , Ltd

更新记录

版本	修改人	修改日期	修改说明	核定人
V1.0.0	AI 组	2024-3-23	初始版本	熊伟

目 录

1 RKLLM 简介

1.1 RKLLM 工具链介绍

1.1.1 RKLLM-Toolkit 功能介绍

RKLLM-Toolkit 是为用户提供在计算机上进行大语言模型的量化、转换的开发套件。通过该工具提供的 Python 接口可以便捷地完成以下功能：

1. 模型转换：支持将 Hugging Face 格式的大语言模型（Large Language Model, LLM）转换为 RKLLM 模型，目前支持的模型包括 LLaMA、Qwen/Qwen2、Phi2 等，转换后的 RKLLM 模型能够在 Rockchip NPU 平台上加载使用。

2. 量化功能：支持将浮点模型量化为定点模型，目前支持的量化类型包括 w4a16 和 w8a8。

1.1.2 RKLLM Runtime 功能介绍

RKLLM Runtime 主要负责加载 RKLLM-Toolkit 转换得到的 RKLLM 模型，并在 RK3576/RK3588 板端通过调用 NPU 驱动在 Rockchip NPU 上实现 RKLLM 模型的推理。在推理 RKLLM 模型时，用户可以自行定义 RKLLM 模型的推理参数设置，定义不同的文本生成方式，并通过预先定义的回调函数不断获得模型的推理结果。

1.2 RKLLM 开发流程介绍

RKLLM 的整体开发步骤主要分为 2 个部分：模型转换和板端部署运行。

1. 模型转换：

在这一阶段，用户提供的 Hugging Face 格式的大语言模型将会被转换为 RKLLM 格式，以便在 Rockchip NPU 平台上进行高效的推理。这一步骤包括：

a. 获取原始模型：获取 Hugging Face 格式的大语言模型；或是自行训练得到的大语言模型，要求模型保存的结构与 Hugging Face 平台上的模型结构一致。

b. 模型加载：通过 rkllm.load_huggingface()函数加载原始模型。

c. 模型量化配置：通过 rkllm.build() 函数构建 RKLLM 模型，在构建过程中可选择是否进行模型量化来提高模型部署在硬件上的性能，以及选择不同的优化等级和量化类型。

d. 模型导出：通过 rkllm.export_rkllm() 函数将 RKLLM 模型导出为一个.rkllm 格式文件，用于后续的部署。

2. 板端部署运行：

这个阶段涵盖了模型的实际部署和运行。它通常包括以下步骤：

a. 模型初始化：加载 RKLLM 模型到 Rockchip NPU 平台，进行相应的模型参数设置来定义所需的文本生成方式，并提前定义用于接受实时推理结果的回调函数，进行推理前准备。

b. 模型推理：执行推理操作，将输入数据传递给模型并运行模型推理，用户可以通过预先定义的回调函数不断获取推理结果。

c. 模型释放：在完成推理流程后，释放模型资源，以便其他任务继续使用 NPU 的计算资源。

这两个步骤构成了完整的 RKLLM 开发流程，确保大语言模型能够成功转换、调试，并最终在 Rockchip NPU 上实现高效部署。

1.3 适用的硬件平台

本文档适用的硬件平台主要包括：RK3576、RK3588。

2 开发环境准备

在发布的 RKLLM 工具链压缩文件中，包含了 RKLLM-Toolkit 的 whl 安装包、RKLLM Runtime 库的相关文件以及参考示例代码，具体的文件夹结构如下：

```
doc
└──Rockchip_RKLLM_SDK_CN.pdf    # RKLLM SDK 说明文档

rkllm-runtime
├──example
│    └── src
│         └── main.cpp
│    └── build-android.sh
│    └── build-linux.sh
│    └── CMakeLists.txt
│    └── Readme.md
├──runtime
│    └── Android
│         └── librkllm_api
│              └──arm64-v8a
│                   └── librkllmrt.so  # RKLLM Runtime 库
│              └──include
│                   └── rkllm.h        # Runtime 头文件
│    └── Linux
│         └── librkllm_api
│              └──aarch64
│                   └── librkllmrt.so
│              └──include
│                   └── rkllm.h

rkllm-toolkit
├──examples
│    └── huggingface
│         └── test.py
├──packages
│    └── md5sum.txt
│    └── rkllm_toolkit-1.0.0-cp38-cp38-linux_x86_64.whl

rknpu-driver
└──rknpu_driver_0.9.6_20240322.tar.bz2
```

在本章中将会对 RKLLM-Toolkit 工具及 RKLLM Runtime 的安装进行详细的介绍，具体的使用方法请参考第 3 章中的使用说明。

2.1 RKLLM-Toolkit 安装

本节主要说明如何通过 pip 方式来安装 RKLLM-Toolkit，用户可以参考以下的具体流程说明完成 RKLLM-Toolkit 工具链的安装。

2.1.1 通过 pip 方式安装

2.1.1.1 安装 miniforge3 工具

为防止系统对多个不同版本的 Python 环境的需求，建议使用 miniforge3 管理 Python 环境。

检查是否安装 miniforge3 和 conda 版本信息，若已安装则可省略此小节步骤。

```
conda -V
# 提示 conda: command not found 则表示未安装 conda
# 提示 例如版本 conda 23.9.0
```

下载 miniforge3 安装包

```
wget -c https://mirrors.bfsu.edu.cn/github-release/conda-
forge/miniforge/LatestRelease/Miniforge3-Linux-x86_64.sh
```

安装 miniforge3

```
chmod 777 Miniforge3-Linux-x86_64.sh
bash Miniforge3-Linux-x86_64.sh
```

2.1.1.2 创建 RKLLM-Toolkit Conda 环境

进入 Conda base 环境

```
source ~/miniforge3/bin/activate # miniforge3 为安装目录
# (base) xxx@xxx-pc:~$
```

创建一个 Python3.8 版本（建议版本）名为 RKLLM-Toolkit 的 Conda 环境

```
conda create -n RKLLM-Toolkit python=3.8
```

进入 RKLLM-Toolkit Conda 环境

```
conda activate RKLLM-Toolkit
# (RKLLM-Toolkit) xxx@xxx-pc:~$
```

2.1.1.3 安装 RKLLM-Toolkit

在 RKLLM-Toolkit Conda 环境下使用 pip 工具直接安装所提供的工具链 whl 包，在安装过程中，安装工具会自动下载 RKLLM-Toolkit 工具所需要的相关依赖包。

```
pip3 install rkllm_toolkit-1.0.0-cp38-cp38-linux_x86_64.whl
```

若执行以下命令没有报错，则安装成功。

```
python
from rkllm.api import RKLLM
```

2.2 RKLLM Runtime 库的使用

在所公开的的 RKLLM 工具链文件中，包括包含 RKLLM Runtime 的全部文件：

- lib/librkllmrt.so: 适用于 RK3576/RK3588 板端调用进行 RKLLM 模型部署推理的 RKLLM Runtime 库；
- include/rkllm_api.h: 与 librkllmrt.so 函数库相对应的头文件，其中包含相关结构体及函数定义的说明；

在通过 RKLLM 工具链构建 RK3576/RK3588 板端的部署推理代码时，需要注意对以上头文件及函数库的链接，从而保证编译的正确性。当代码在 RK3576/RK3588 板端实际运行的过程中，同样需要确保以上函数库文件成功推送至板端，并通过以下环境变量设置完成函数库的声明：

```
export LD_LIBRARY_PATH=/path/to/your/lib
```

2.3 RKLLM Runtime 的编译要求

在使用 RKLLM Runtime 的过程中，需要注意 gcc 编译器的版本问题。推荐使用交叉编译工具 gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu；具体的下载路径为：GCC 10.2 交叉编译工具下载地址。请注意，交叉编译工具往往向下兼容而无法向上兼容，因此不要使用 10.2 以下的版本。

若是选择使用 Android 平台，需要进行 Android 可执行文件的编译，推荐使用 Android NDK 工具进行交叉编译，下载路径为：Android NDK 交叉编译工具下载地址，推荐使用 r18b 版本。

具体的编译方式也可以参考 RKLLM-Toolkit 工具链文件中的 example/build_demo.sh。

2.4 芯片内核更新

由于当前公开的固件内核驱动版本不支持 RKLLM 工具，因此需要更新内核。rknpu 驱动包支持两个主要内核版本：kernel-5.10 和 kernel-6.1。对于 kernel-5.10，建议使用具体版本号 5.10.198，repo：GitHub - rockchip-linux/kernel at develop-5.10；对于 kernel-6.1，建议使用具体版本号 6.1.57。可在内核根目录下的 Makefile 中确认具体版本号。

更新步骤如下：

a. 下载压缩包 rknpu_driver_0.9.6_20240322.tar.bz2。

b. 解压该压缩包，将其中的 rknpu 驱动代码覆盖到当前内核代码目录。

c. 重新编译内核。

d. 将新编译的内核烧录到设备中。

3 RKLLM 使用说明

3.1 模型转换

RKLLM-Toolkit 提供模型的转换、量化功能。作为 RKLLM-Toolkit 的核心功能之一，它允许用户将 Hugging Face 格式的大语言模型转换为 RKLLM 模型，从而将 RKLLM 模型在 Rockchip NPU 上加载运行。本节将重点介绍 RKLLM-Toolkit 对 LLM 模型的具体转换实现，以供用户参考。

3.1.1 RKLLM 初始化

在这一部分，用户需要先初始化 RKLLM 对象，这是整个工作流的第一步。在示例代码中使用 RKLLM()构造函数来初始化 RKLLM 对象：

```
rkllm = RKLLM()
```

3.1.2 模型加载

在 RKLLM 初始化完成后，用户需要调用 rkllm.load_huggingface()函数来传入模型的具体路径，RKLLM-Toolkit 即可根据对应路径顺利加载 Hugging Face 格式的大语言模型，从而顺利完成后续的转换、量化操作，具体的函数定义如下：

表 3-1 load_huggingface 函数接口说明

函数名	load_huggingface
描述	用于加载开源的 Hugging Face 格式的大语言模型。
参数	***model:*** LLM 模型文件的存放路径，用于加载模型进行后续的转换、量化；
返回值	**0** 表示模型加载正常；**-1** 表示模型加载失败；

示例代码如下：

```
ret = rkllm.load_huggingface(model = './qwen')
if ret != 0:
    print('Load model failed!')
```

3.1.3 RKLLM 模型的量化构建

用户在通过 rkllm.load_huggingface()函数完成原始模型的加载后，下一步就是通过 rkllm.build()

函数实现对 RKLLM 模型的构建。构建模型时，用户可以选择是否进行量化，量化有助于减小模型的大小和提高在 Rockchip NPU 上的推理性能。rkllm.build()函数的具体定义如下：

表 3-2 build 函数接口说明

函数名	build
描述	用于构建得到 RKLLM 模型，并在转换过程中定义具体的量化操作。
参数	***do_quantization:*** 该参数控制是否对模型进行量化操作，建议设置为 True； ***optimization_level:*** 该参数用于设置模型权重的优化等级，目前的优化等级支持设置为{0, 1}。optimization_level=0 表示不进行任何优化操作，optimization_level=1 表示对原始权重进行优化以提高量化精度； ***quantized_dtype:*** 该参数用于设置量化的具体类型，目前支持的量化类型包括“w4a16”和“w8a8”，“w4a16”表示对权重进行 4bit 量化而对激活值不进行量化；“w8a8”表示对权重和激活值均进行 8bit 量化；目前 rk3576 平台支持“w4a16”和“w8a8”两种量化类型，rk3588 仅支持“w8a8”量化类型； ***target_platform:*** 模型运行的硬件平台，可选择的设置包括“rk3576”或“rk3588”；
返回值	**0** 表示模型转换、量化正常；**-1** 表示模型转换失败；

示例代码如下：

```
ret = rkllm.build(
      do_quantization=True,
      optimization_level=1,
      quantized_dtype='w8a8',
      target_platform='rk3588')
if ret != 0:
   print('Build model failed!')
```

3.1.4 导出 RKLLM 模型

用户在通过 rkllm.build()函数构建了 RKLLM 模型后，可以通过 rkllm.export_rkllm()函数将 RKNN 模型保存为一个.rkllm 文件，以便后续模型的部署。rkllm.export_rkllm()函数的具体参数定义如下：

表 3-3 export_rkllm 函数接口说明

函数名	export_rkllm
描述	用于保存转换、量化后的 RKLLM 模型，用于后续的推理调用。
参数	***export_path:*** 导出 RKLLM 模型文件的保存路径；
返回值	**0** 表示模型成功导出保存；**-1** 表示模型导出失败；

示例代码如下：

```
ret = rkllm.export_rkllm(export_path = './qwen_w8a8.rkllm')
if ret != 0:
    print('Export model failed!')
```

以上的这些操作涵盖了 RKLLM-Toolkit 模型转换、量化的全部步骤，根据不同的需求和应用场景，用户可以选择不同的配置选项和量化方式进行自定义设置，方便后续进行部署。

3.2 板端推理实现

此章节介绍通用 API 接口函数的调用流程，用户可以参考本节内容使用 C++构建代码，在板端实现对 RKLLM 模型的推理，获取推理结果。RKLLM 板端推理实现的调用流程如下：

1. 定义回调函数 callback()；
2. 定义 RKLLM 模型参数结构体 RKLLMParam；
3. rkllm_init()初始化 RKLLM 模型；
4. rkllm_run()进行模型推理；
5. 通过回调函数 callback()对模型实时传回的推理结果进行处理；
6. rkllm_destroy()销毁 RKLLM 模型并释放资源；

在本节的后续部分，该文档将详细介绍流程的各环节，并对其中的函数进行详细说明。

3.2.1 回调函数定义

回调函数是用于接收模型实时输出的结果。在初始化 RKLLM 的过程中，回调函数将被绑定，随着模型推理的进行不断将结果输出至回调函数中。

示例代码如下，该回调函数将输出的文本结果实时地打印输出到终端中：

```
void callback(const char* text, void* userdata, LLMCallState state)
{
    if(state == LLM_RUN_NORMAL){
        printf("%s", text);
    }
    if (state == LLM_RUN_FINISH) {
        printf("finish\n");
    } else if (state == LLM_RUN_ERROR){
        printf("\run error\n");
    }
}
```

其中，LLMCallState 是一个状态标志，其具体定义如下：

表 3-4 LLMCallState 状态标志说明

变量名	LLMCallState
描述	用于表示当前 RKLLM 的运行状态。
参数	无
返回值	**0, *LLM_RUN_NORMAL,*** 表示 RKLLM 模型当前正在推理中； **1, *LLM_RUN_FINISH,*** 表示 RKLLM 模型已完成当前输入的全部推理； **2, *LLM_RUN_ERROR,*** 表示 RKLLM 模型推理出现错误；

用户在回调函数的设计过程中，可以根据 LLMCallState 的不同状态设置不同的后处理行为；

3.2.2 参数结构体 RKLLMParam 定义

结构体 RKLLMParam 用于描述、定义 RKLLM 的详细信息，具体的定义如下：

表 3-5 RKLLMParam 结构体参数说明

变量名	RKLLMParam
描述	用于定义 RKLLM 模型的各项细节参数。
参数	***const char* modelPath:*** 模型文件的存放路径； ***const char* target_platform:*** 模型运行的硬件平台，可选择的设置包括“rk3576”或“rk3588”； ***int32_t num_npu_core:*** 模型推理时使用的 NPU 核心数量，若 target_platform 设置为“rk3576”，可配置的范围为[1, 2]；“rk3588”可配置的范围则为[1, 3]；

	int32_t max_context_len: 设置 prompt 的上下文大小 ***int32_t max_new_tokens:*** 用于设置模型推理时生成 Token 的数量上限； ***int32_t top_k:*** top-k 采样是一种文本生成方法，它仅从模型预测的前 k 个最可能的 Token 中选择下一个 Token。这种方法有助于减少生成低概率或无意义 Token 的风险。更高的 top-k 值（如 100）将考虑更多的 Token 选择，导致文本更加多样化；而更低的值（如 10）将聚焦于最可能的 Token，生成更加保守的文本。默认值为 40； ***float top_p:*** top-p 采样，也被称为核心采样，是另一种文本生成方法，从累计概率至少为 p 的一组 Token 中选择下一个 Token。这种方法通过考虑 Token 的概率和采样的 Token 数量来在多样性和质量之间提供平衡。更高的 top-p 值（如 0.95）会使生成的文本更加多样化；而更低的值（如 0.5）将生成更加集中和保守的文本。默认值为 0.9； ***float temperature:*** 控制生成文本随机性的超参数，它通过调整模型输出 Token 的概率分布来发挥作用；更高的温度（如 1.5）会使输出更加随机和创造性，当温度较高时，模型在选择下一个 Token 时会考虑更多可能性较低的选项，从而产生更多样和意想不到的输出；更低的温度（例 0.5）会使输出更加集中、保守，较低的温度意味着模型在生成文本时更倾向于选择概率高的 Token，从而导致更一致、更可预测的输出；温度为 0 的极端情况下，模型总是选择最有可能的下一个 Token，这会导致每次运行时输出完全相同；为了确保随机性和确定性之间的平衡，使输出既不过于单一和可预测，也不过于随机和杂乱，默认值为 0.8； ***float repeat_penalty:*** 控制生成文本中 Token 序列重复的情况，帮助防止模型生成重复或单调的文本。更高的值（例如 1.5）将更强烈地惩罚重复，而较低的值（例如 0.9）则更为宽容。默认值为 1.1； ***float frequency_penalty:*** 单词/短语重复度惩罚因子，减少总体上使用频率较高的单词/短语的概率，增加使用频率较低的单词/短语的可能性，这可能会使生成的文本

	更加多样化，但也可能导致生成的文本难以理解或不符合预期。设置范围为[-2.0, 2.0]，默认为 0； ***int32_t mirostat:*** 在文本生成过程中主动维持生成文本的质量在期望的范围内的算法，它旨在在连贯性和多样性之间找到平衡，避免因过度重复（无聊陷阱）或不连贯（混乱陷阱）导致的低质量输出；取值空间为{0, 1, 2}, 0 表示不启动该算法，1 表示使用 mirostat 算法，2 则表示使用 mirostat2.0 算法； ***float mirostat_tau:*** 选项设置 mirostat 的目标熵，代表生成文本的期望困惑度值。调整目标熵可以让你控制生成文本中连贯性与多样性的平衡。较低的值将导致文本更加集中和连贯，而较高的值将导致文本更加多样化，可能连贯性较差。默认值是 5.0； ***float mirostat_eta:*** 选项设置 mirostat 的学习率，学习率影响算法对生成文本反馈的响应速度。较低的学习率将导致调整速度较慢，而较高的学习率将使算法更加灵敏。默认值是 0.1；
返回值	无

在实际的代码构建中，RKLLMParam 需要调用 rkllm_createDefaultParam()函数来初始化定义，并根据需求设置相应的模型参数。示例代码如下：

```
RKLLMParam param = rkllm_createDefaultParam();
param.modelPath = "models/qwen-1.8b/qwen_w8a8.rkllm";
param.target_platform = "rk3588";
param.num_npu_core = 3;
param.top_k = 1;
param.max_new_tokens = 256;
param.max_context_len = 512;
```

3.2.3 初始化模型

在进行模型的初始化之前，需要提前定义 LLMHandle 句柄，该句柄用于模型的初始化、推理和资源释放过程。注意，只有统一这 3 个流程中的 LLMHandle 句柄对象，才能够正确完成模型的推理流程。

在模型推理前，用户需要通过 rkllm_init()函数完成模型的初始化，具体函数的定义如下：

表 3-6 rkllm_init 函数接口说明

函数名	rkllm_init
描述	用于初始化 RKLLM 模型的具体参数及相关推理设置。
参数	***LLMHandle* handle:*** 将模型注册到相应句柄中，用于后续推理、释放调用； ***RKLLMParam param:*** 模型定义的参数结构体，详见 3.2.2 参数结构体 RKLLMParam 定义； ***LLMResultCallback callback:*** 用于接受处理模型实时输出的回调函数，详见 3.2.1 回调函数定义；
返回值	**0** 表示初始化流程正常；**-1** 表示初始化失败；

示例代码如下：

```
LLMHandle llmHandle = nullptr;
rkllm_init(&llmHandle, param, callback);
```

3.2.4 模型推理

用户在完成 RKLLM 模型的初始化流程后，即可通过 rkllm_run()函数进行模型推理，并可以通过初始化时预先定义的回调函数对实时推理结果进行处理；rkllm_run()的具体函数定义如下：

表 3-7 rkllm_run 函数接口说明

函数名	rkllm_run
描述	调用完成初始化的 RKLLM 模型进行结果推理；
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄，可见 3.2.3 初始化模型； ***const char* prompt:*** 模型推理的输入 prompt，即用户的“问题”，注意需要在问题前后加上预设的 prompt，保证推理正确性，详见示例代码； ***void* userdata:*** 用户自定义的函数指针，默认设置为 NULL 即可；
返回值	**0** 表示模型推理正常运行；**-1** 表示调用模型推理失败；

模型推理的示例代码如下：

```
// 提前定义 prompt 前后的文本预设值
#define PROMPT_TEXT_PREFIX "<|im_start|>system You are a helpful
assistant. <|im_end|> <|im_start|>user"
#define PROMPT_TEXT_POSTFIX "<|im_end|><|im_start|>assistant"

// 定义输入的 prompt 并完成前后 prompt 的拼接
string input_str = "把这句话翻译成英文：RK3588 是新一代高端处理器，具有高算力、
低功耗、超强多媒体、丰富数据接口等特点";
input_str = PROMPT_TEXT_PREFIX + input_str + PROMPT_TEXT_POSTFIX;

// 调用模型推理接口,其中 llmHandle 为模型初始化时传入的句柄
rkllm_run(llmHandle, input_str.c_str(), NULL);
```

3.2.5 释放模型资源

在完成全部的模型推理调用后，用户需要调用 rkllm_destroy()函数进行 RKLLM 模型的销毁，并释放所申请的 CPU、NPU 计算资源，以供其他进程、模型的调用。具体的函数定义如下：

表 3-8 rkllm_ destroy 函数接口说明

函数名	rkllm_ destroy
描述	用于销毁 RKLLM 模型并释放所有计算资源。
参数	***LLMHandle handle:*** 模型初始化注册的目标句柄，可见 3.2.3 初始化模型；
返回值	**0** 表示 RKLLM 模型正常销毁、释放；**-1** 表示模型释放失败；

示例代码如下：

```
// 其中 llmHandle 为模型初始化时传入的句柄
rkllm_destroy(llmHandle);
```

3.3 板端推理调用的完整示例

以下为一份板端推理调用的完整 C++代码示例，与工具链文件中的 example/mian.cpp 相同，该实现了包括模型初始化、模型推理、回调函数处理输出和模型资源释放等全部流程，用户可以参考相关代码进行自定义功能的实现。

```
// Copyright (c) 2024 by Rockchip Electronics Co., Ltd. All Rights
Reserved.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//    http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#include <string.h>
#include <unistd.h>
#include <string>
#include "rkllm.h"
#include <fstream>
#include <iostream>
#include <csignal>
#include <vector>

#define PROMPT_TEXT_PREFIX "<|im_start|>system You are a helpful
assistant. <|im_end|> <|im_start|>user"
#define PROMPT_TEXT_POSTFIX "<|im_end|><|im_start|>assistant"

using namespace std;
LLMHandle llmHandle = nullptr;

void exit_handler(int signal)
{
    if (llmHandle != nullptr)
    {
        {
            cout << "程序即将退出" << endl;
            LLMHandle _tmp = llmHandle;
            llmHandle = nullptr;
            rkllm_destroy(_tmp);
        }
        exit(signal);
    }
}

void callback(const char *text, void *userdata, LLMCallState state)
{

    if (state == LLM_RUN_FINISH)
    {
        printf("\n");
    }
    else if (state == LLM_RUN_ERROR)
    {
        printf("\\run error\n");
    }
    else{
```

```
        printf("%s", text);
    }
}

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        printf("Usage:%s [rkllm_model_path]\n", argv[0]);
        return -1;
    }
    signal(SIGINT, exit_handler);
    string rkllm_model(argv[1]);
    printf("rkllm init start\n");

    //设置参数及初始化
    RKLLMParam param = rkllm_createDefaultParam();
    param.modelPath = rkllm_model.c_str();
    param.target_platform = "rk3588";
    param.num_npu_core = 3;
    param.top_k = 1;
    param.max_new_tokens = 256;
    param.max_context_len = 512;
    rkllm_init(&llmHandle, param, callback);
    printf("rkllm init success\n");

    vector<string> pre_input;
    pre_input.push_back("把下面的现代文翻译成文言文：到了春风和煦，阳光明媚的时候，湖面平静，没有惊涛骇浪，天色湖光相连，一片碧绿，广阔无际；沙洲上的鸥鸟，时而飞翔，时而停歇，美丽的鱼游来游去，岸上与小洲上的花草，青翠欲滴。");
    pre_input.push_back("以咏梅为题目，帮我写一首古诗，要求包含梅花、白雪等元素。");
    pre_input.push_back("上联：江边惯看千帆过");
    pre_input.push_back("把这句话翻译成英文：RK3588 是新一代高端处理器，具有高算力、低功耗、超强多媒体、丰富数据接口等特点");
    cout << "\n**********************可输入以下问题对应序号获取回答/或自定义输入********************\n"
         << endl;
    for (int i = 0; i < (int)pre_input.size(); i++)
    {
        cout << "[" << i << "] " << pre_input[i] << endl;
    }
    cout <<
"\n**************************************************************************\n"
         << endl;

    string text;
    while (true)
    {
        std::string input_str;
        printf("\n");
        printf("user: ");
        std::getline(std::cin, input_str);
        if (input_str == "exit")
        {
            break;
```

```
        }
        for (int i = 0; i < (int)pre_input.size(); i++)
        {
            if (input_str == to_string(i))
            {
                input_str = pre_input[i];
                cout << input_str << endl;
            }
        }
        string text = PROMPT_TEXT_PREFIX + input_str +
PROMPT_TEXT_POSTFIX;
        printf("robot: ");
        rkllm_run(llmHandle, text.c_str(), NULL);
    }

    rkllm_destroy(llmHandle);

    return 0;
}
```

3.4 示例代码的使用说明

在工具链压缩文件夹中包含了 example 目录，该目录下包含了完整的 RKLLM 模型推理调用的示例代码 example/main.cpp，同时包含了编译脚本 example/build_demo.sh。本节将会对示例代码的使用进行简要说明。

首先，用户需要自行准备交叉编译工具，注意在 2.3 中说明的编译工具版本推荐为 10.2 及以上，随后在编译过程前自行替换 example/build_demo.sh 中的交叉编译工具路径：

```
# 设置交叉编译器路径为本地工具所在路径
# GCC_COMPILER_PATH=aarch64-linux-gnu
GCC_COMPILER_PATH=~/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-
gnu/bin/aarch64-none-linux-gnu
```

编译完成后，用户即可通过运行 example/build_demo.sh 在 example 路径下构建得到板端可执行文件 llm_demo。随后需要将可执行文件、函数库文件夹 lib 及 RKLLM 模型（提前使用 RKLLM-Toolkit 工具完成转换、量化）推送至板端：

```
# 假设当前位于 rkllm-toolkit 的主目录下
adb push ./example/llm_demo /path/to/your/board/workshop
adb push ./lib /path/to/your/board/workshop
adb push ./xxx.rkllm /path/to/your/board/workshop
```

在完成以上步骤后，用户即可通过 adb 进入板端终端界面，并进入相应的 workshop 文件夹目录下，通过以下指令进行 RKLLM 的板端推理调用：

```
# 假设板端上的 workshop 目录下
export LD_LIBRARY_PATH=/path/to/your/lib    # 通过环境变量指定函数库路径
./llm_demo ./xxx.rkllm          # 可执行文件在执行时需要传入 rkllm 的具体路径
```

通过以上操作，用户即可进入示例推理界面，与板端模型进行推理交互，并实时获取 RKLLM 的推理结果。

4 相关资源

RKLLM：https://github.com/airockchip/rknn-llm

RKNN：https://github.com/airockchip/rknn-toolkit2