Cambricon CNRTC 用户手册

版本 *0.6.0*

2022 年 08 月 15 日

Cambricon
寒 武 纪

目录

插图目录

1 版权声明

免责声明

中科寒武纪科技股份有限公司（下称“寒武纪”）不代表、担保（明示、暗示或法定的）或保证本文件所含信息，并明示放弃对可销售性、所有权、不侵犯知识产权或特定目的适用性做出任何和所有暗示担保，且寒武纪不承担因应用或使用任何产品或服务而产生的任何责任。寒武纪不应对因下列原因产生的任何违约、损害赔偿、成本或问题承担任何责任：（1）使用寒武纪产品的任何方式违背本指南；或（2）客户产品设计。

责任限制

在任何情况下，寒武纪都不对因使用或无法使用本指南而导致的任何损害（包括但不限于利润损失、业务中断和信息损失等损害）承担责任，即便寒武纪已被告知可能遭受该等损害。尽管客户可能因任何理由遭受任何损害，根据寒武纪的产品销售条款与条件，寒武纪为本指南所述产品对客户承担的总共和累计责任应受到限制。

信息准确性

本文件提供的信息属于寒武纪所有，且寒武纪保留不经通知随时对本文件信息或对任何产品和服务做出任何更改的权利。本指南所含信息和本指南所引用寒武纪文档的所有其他信息均“按原样”提供。寒武纪不担保信息、文本、图案、链接或本指南内所含其他项目的准确性或完整性。寒武纪可不经通知随时对本指南或本指南所述产品做出更改，但不承诺更新本指南。

本指南列出的性能测试和等级要使用特定芯片或计算机系统或组件来测量。经该等测试，本指南所示结果反映了寒武纪产品的大概性能。系统硬件或软件设计或配置的任何不同会影响实际性能。如上所述，寒武纪不代表、担保或保证本指南所述产品将适用于任何特定用途。寒武纪不代表或担保测试每种产品的所有参数。客户全权承担确保产品适合并适用于客户计划的应用以及对应用程序进行必要测试的责任，以期避免应用程序或产品的默认情况。

客户产品设计的脆弱性会影响寒武纪产品的质量和可靠性并导致超出本指南范围的额外或不同的情况和/或要求。

知识产权通知

寒武纪和寒武纪的标志是中科寒武纪科技股份有限公司在中国和其他国家的商标和/或注册商标。其他公司和产品名称应为与其关联的各自公司的商标。

式复制、重制、修改、出版、上传、发布、传输或分发本指南。除了客户使用本指南信息和产品的权利，根据本指南，寒武纪不授予其他任何明示或暗示的权利或许可。未免疑义，寒武纪不根据任何专利、版权、商标、商业秘密或任何其他寒武纪的知识产权或所有权对客户授予任何（明示或暗示的）权利或许可。

- 版权声明
- © 2022 中科寒武纪科技股份有限公司保留一切权利。

2 更新历史

- **V0.6.0**

  更新时间：2022 年 7 月 14 日

  更新内容：

  – 修改 samples 里 cnInvokeKernel 函数的参数传递方式。

- **V0.5.2**

  更新时间：2022 年 7 月 1 日

  更新内容：

  – 增加 samples 里的 mtp_592 编译选项，支持在 mtp_592 设备上的编译和运行。

- **V0.5.1**

  更新时间：2022 年 2 月 23 日

  更新内容：

  – 增强功能:

    * 消除cnrtcCompileCode 函数返回错误值时的内存泄露风险。

- **V0.5.0**

  更新时间：2021 年 6 月 29 日

  更新内容：

  – 新增 API 接口:

    * cnrtcCreateCodeV2

  – 新增数据类型:

    * cnrtcCodeType

  – 新增功能:

    * 支持编译 MLISA 代码。

- **V0.4.0**

  更新时间：2021 年 6 月 11 日

  更新内容：

  – 新增 API 接口:

    * cnrtcGetCompilationOutput
    * cnrtcGetCompilationOutputSize

  – 修改 API 接口:

    * cnrtcStatus
    * cnrtcGetFatBinary

    * cnrtcGetFatBinarySize

- **V0.3.1**

  更新时间：2021 年 3 月 26 日

  更新内容：

  – 示例代码修改 cnMemcpy 函数参数。

- **V0.3.0**

  更新时间：2021 年 3 月 4 日

  更新内容：

  – 新增FAQ 章节。

- **V0.2.1**

  更新时间：2021 年 2 月 22 日

  更新内容：

  – 前言

  – API 接口:

    * cnrtcCompileCode

- **V0.2.0**

  更新时间：2020 年 12 月 21 日

  更新内容：

  – 前言

  – 运行环境

  – API 接口:

    * cnrtcCompileCode

- **V0.1.0**

  更新时间：2020 年 10 月 24 日

  更新内容：

  – 版权声明

  – 前言

  – 概述

  – 环境依赖

  – 实例代码

  – 编程模型

  – API 接口:

    * cnrtcStatus
    * cnrtcTransStatusToString
    * cnrtcVersion
    * cnrtcCode
    * cnrtcCreateCode
    * cnrtcDestroyCode
    * cnrtcCompileCode

* cnrtcGetFatBinary
* cnrtcGetFatBinarySize
* cnrtcGetCompilationLog
* cnrtcGetCompilationLogSize

3 概述

Cambricon CNRTC（Cambricon Runtime Compilation Library，寒武纪运行时编译库）是 Cambricon BANG C 和 MLISA 语言的运行时编译库，可以接收字符串形式的 Cambricon BANG C 或 MLISA 代码，即时编译为 `cnFatBinary` 二进制，并通过驱动程序解析 `cnFatBinary` 后运行。

- Cambricon CNRTC 提供 Cambricon BANG C 和 MLISA 代码的即时编译功能。
- Cambricon CNRTC 支持多种编译优化选项。
- Cambricon CNRTC 支持多款寒武纪硬件产品。
- Cambricon CNRTC 支持输出编译错误信息。
- Cambricon CNRTC 编译后的二进制可以配合驱动程序（CNDrv）在寒武纪硬件设备上运行。
- Cambricon CNRTC 保证线程安全。只要用户根据 `cnrtcCode` 对象的使用要求，多个线程不共享同一个 `cnrtcCode` 对象，则可以保证各函数从多个主机线程独立调用。

4 运行环境

4.1 环境依赖

Cambricon CNRTC 依赖的软硬件环境如下：

- 操作系统：Linux x86_64。
- MLU：MLU220, MLU270, MLU290, CE3226, MLU370。
- CNCC（Cambricon Compiler Collection，寒武纪 Cambricon BANG C 语言编译器）v4.0.0 及以上版本。
- CNAS（Cambricon Assembler，寒武纪 MLISA 语言汇编器）v4.0.0 及以上版本。
- CNBIN（Cambricon Binary，寒武纪二进制库）v1.0.0 及以上版本。
- CNDrv（Cambricon Driver API，寒武纪软件栈驱动接口）v2.0.0 及以上版本。

4.2 目录结构

上述依赖文件以及 Cambricon CNRTC 的头文件和动态链接库需要按下面的目录结构组织：

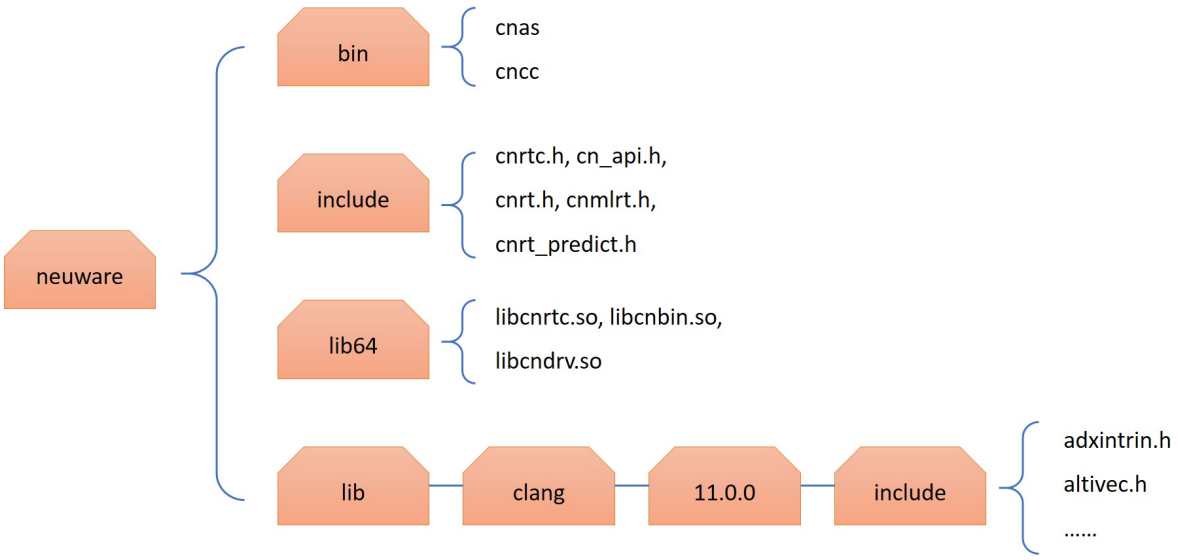

图 4.1: 目录结构

5 编程模型

Cambricon CNRTC 的编程模型，就是“Cambricon BANG C/MLISA 源代码——cnFatBinary 二进制——CNmodule 模型——CNkernel 核函数”的转换过程。如图所示：

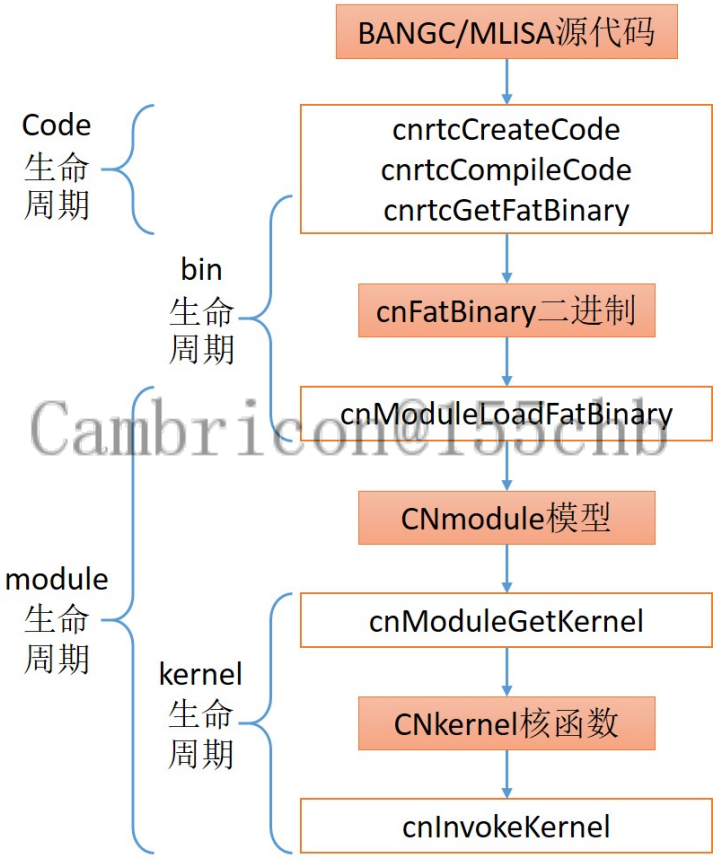

图 5.1: 编程模型

6 数据类型

6.1 cnrtcStatus

```
enum cnrtcStatus
```

枚举类cnrtcStatus 定义了多种函数返回值，便于用户获知函数的执行状态并自查错误。

取值及说明

- CNRTC_SUCCESS = 0
  函数运行成功。
- CNRTC_VERSION_NO_SPACE_ERROR = 1
  cnrtcVersion 函数的传入参数未分配空间。
- CNRTC_CREATE_CODE_ALREADY_EXIST_ERROR = 2
  cnrtcCreateCode 函数接收的cnrtcCode 对象已经存在，本次调用属于重复创建cnrtcCode 对象。
- CNRTC_CREATE_CODE_SOURCE_NULL_ERROR = 3
  cnrtcCreateCode 函数未传入源代码。
- CNRTC_DESTROY_CODE_DOUBLE_FREE_ERROR = 4
  cnrtcDestroyCode 函数传入的cnrtcCode 对象已经销毁，本次调用属于重复销毁。
- CNRTC_COMPILE_CODE_NO_CODE_ERROR = 5
  cnrtcCompileCode 函数传入的cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_COMPILE_CODE_PARAM_ERROR = 6
  cnrtcCompileCode 函数传入的编译选项参数错误。
- CNRTC_COMPILE_CODE_WRITE_PERMISSION_ERROR = 7
  cnrtcCompileCode 函数不能获取系统的写权限。
- CNRTC_COMPILE_CODE_READ_PERMISSION_ERROR = 8
  cnrtcCompileCode 函数不能获取系统的读权限。
- CNRTC_COMPILE_CODE_ENVIRONMENT_ERROR = 9
  cnrtcCompileCode 函数运行环境异常。
- CNRTC_COMPILE_CODE_ERROR = 10
  cnrtcCompileCode 函数遇到源代码编译错误，请检查源代码。
- CNRTC_GET_BIN_NO_CODE_ERROR = 11
  cnrtcGetFatBinary 函数传入的cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。

- CNRTC_GET_BIN_NO_SPACE_ERROR = 12
  cnrtcGetFatBinary 函数接收二进制的内存指针未分配空间。
- CNRTC_GET_BIN_SIZE_NO_CODE_ERROR = 13
  cnrtcGetFatBinarySize 函数传入的cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_GET_BIN_SIZE_NO_SPACE_ERROR = 14
  cnrtcGetFatBinarySize 函数接收二进制内存空间大小的内存指针未分配空间。
- CNRTC_GET_LOG_NO_CODE_ERROR = 15
  cnrtcGetCompilationLog 函数传入的cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_GET_LOG_NO_SPACE_ERROR = 16
  cnrtcGetCompilationLog 函数传入的内存指针参数未分配空间。
- CNRTC_GET_LOG_SIZE_NO_CODE_ERROR = 17
  cnrtcGetCompilationLogSize 函数传入的cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_GET_LOG_SIZE_NO_SPACE_ERROR = 18
  cnrtcGetCompilationLogSize 函数接收编译错误记录内存空间大小的内存指针未分配空间。
- CNRTC_NO_CODE_ERROR = 19
  函数接收的cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_NO_OUTPUT_SPACE_ERROR = 20
  函数接收输出数据的内存指针未分配空间。
- CNRTC_NO_FATBIN_OPTION_ERROR = 21
  cnrtcGetFatBinary 或cnrtcGetFatBinarySize 函数在调用前，没有给cnrtcCompileCode 函数添加 `--bang-fatbin-only` (针对 Cambricon BANG C 语言) 或 `--fatbin` (针对 MLISA 语言) 编译选项。

6.2 cnrtcCodeType

```
enum cnrtcCodeType
```

枚举类cnrtcCodeType 定义了 Cambricon CNRTC 支持的输入语言。

取值及说明

- CNRTC_CODE_BANGC = 0
  Cambricon BANG C 语言。
- CNRTC_CODE_MLISA = 1
  MLISA 语言。

6.3 cnrtcCode

```
typedef struct _cnrtcCode *cnrtcCode
```

结构体指针cnrtcCode 是 Cambricon BANG C 或 MLISA 源代码在 Cambricon CNRTC 库中的一个抽象，它是整个程序的主体，其他功能函数都围绕它展开（例如创建、销毁、编译等功能）。

注意

- cnrtcCode 对象的生命周期从调用cnrtcCreateCode 函数开始，到调用cnrtcDestroyCode 函数结束。
- 在初始化时要把它赋值为 NULL 或 nullptr，例如：

```
cnrtcCode code1 = NULL;
```

- 在使用完成后要调用cnrtcDestroyCode 函数将它销毁，否则会内存泄漏。

7 API 接口

7.1 cnrtcTransStatusToString

```
const char* cnrtcTransStatusToString(cnrtcStatus stat)
```

本函数将枚举类cnrtcStatus 转换为字符串输出。例如输入 CNRTC_SUCCESS ，则输出为“CNRTC_SUCCESS”。

参数说明

- stat[in]: 枚举类cnrtcStatus 的任意取值。

返回值描述

- 字符串形式的cnrtcStatus。若输入cnrtcStatus 取值不是 cnrtc.h 头文件所列的值，则返回“Unknown cnrtcStatus”。

7.2 cnrtcVersion

```
cnrtcStatus cnrtcVersion(int *major, int *minor)
```

本函数用于返回 Cambricon CNRTC 库的版本号，以便用户查询。

参数说明

- major[out]: 主版本号。
- minor[out]: 副版本号。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_VERSION_NO_SPACE_ERROR
  传入参数未分配空间。

7.3 cnrtcCreateCode

```
cnrtcStatus cnrtcCreateCode(cnrtcCode* code,
                            const char* srcCode,
                            const char* name,
                            int headerNum,
                            const char** headers,
                            const char** includeNames);
```

本函数用于创建cnrtcCode 对象，接收用户的 Cambricon BANG C 源代码。

参数说明

- code[in]: cnrtcCode 对象。
- srcCode[in]: Cambricon BANG C 源代码。
- name[in]: 预留参数，暂不使用。
- headerNum[in]: 预留参数，暂不使用。
- headers[in]: 预留参数，暂不使用。
- includeNames[in]: 预留参数，暂不使用。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_CREATE_CODE_ALREADY_EXIST_ERROR
  cnrtcCode 对象已经存在，本次调用属于重复创建cnrtcCode 对象。
- CNRTC_CREATE_CODE_SOURCE_NULL_ERROR
  未传入源代码。

注意事项

- srcCode 仅支持 Cambricon BANG C 源代码，如需输入 MLISA，请使用cnrtcCreateCodeV2 函数。
- srcCode 的内存空间由用户创建，但是在传入cnrtcCreateCode 函数后，不可以释放 srcCode 原来的内存空间，只有调用cnrtcDestroyCode 函数后，才可以释放。
- 在cnrtcCode 对象的生命周期内（从创建到销毁），srcCode 不可以改变。
- srcCode 的 Cambricon BANG C 源代码需要遵循 Cambricon BANG C 语言规范，详见《Cambricon BANG C Developer Guide》。
- srcCode 需要在前部引用头文件：`#include "bang.h"` 或 `#include "mlu.h"` 。
- srcCode 主入口函数名前需要添加前缀 `extern "C" __mlu_global__` 或 `extern "C" __mlu_entry__` ，其余函数需要添加前缀 `__mlu_func__` （代表内联函数）或 `__mlu_device__` （代表调用函数）。
- srcCode 不支持 Cambricon BANG C 的混合编程模式，即 CPU 代码和 MLU 代码写在一起。
- srcCode 不支持 Cambricon BANG C 的 `assert()` 函数。

7.4 cnrtcCreateCodeV2

```
cnrtcStatus cnrtcCreateCodeV2(cnrtcCode* code,
                              const char* srcCode,
                              cnrtcCodeType codeType,
                              const char* name,
                              int headerNum,
                              const char** headers,
                              const char** includeNames);
```

本函数用于创建cnrtcCode 对象，接收用户的 Cambricon BANG C 或 MLISA 源代码。

参数说明

- code[in]: cnrtcCode 对象。
- srcCode[in]: Cambricon BANG C 或 MLISA 源代码。
- codeType[in]: 源代码类型。
- name[in]: 预留参数，暂不使用。
- headerNum[in]: 预留参数，暂不使用。
- headers[in]: 预留参数，暂不使用。
- includeNames[in]: 预留参数，暂不使用。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_CREATE_CODE_ALREADY_EXIST_ERROR
  cnrtcCode 对象已经存在，本次调用属于重复创建cnrtcCode 对象。
- CNRTC_CREATE_CODE_SOURCE_NULL_ERROR
  未传入源代码。

注意事项

- srcCode 支持 Cambricon BANG C 和 MLISA 两种语言，请将 codeType 参数设置为相应的语言。
- srcCode 的内存空间由用户创建，但是在传入cnrtcCreateCode 函数后，不可以释放 srcCode 原来的内存空间，只有调用cnrtcDestroyCode 函数后，才可以释放。
- 在cnrtcCode 对象的生命周期内（从创建到销毁），srcCode 不可以改变。
- 若 srcCode 是 Cambricon BANG C 代码，则需要满足以下约束：
  - srcCode 的 Cambricon BANG C 源代码需要遵循 Cambricon BANG C 语言规范，详见《Cambricon BANG C Developer Guide》。
  - srcCode 需要在前部引用头文件：`#include "bang.h"` 或 `#include "mlu.h"` 。
  - srcCode 主入口函数名前需要添加前缀 `extern "C" __mlu_global__` 或 `extern "C" __mlu_entry__` ，其余函数需要添加前缀 `__mlu_func__` （代表内联函数）或 `__mlu_device__`

（代表调用函数）。

- srcCode 不支持 Cambricon BANG C 的混合编程模式，即 CPU 代码和 MLU 代码写在一起。
- srcCode 不支持 Cambricon BANG C 的 `assert()` 函数。

- 若 srcCode 是 MLISA 代码，则需要满足以下约束：
  - srcCode 的 MLISA 源代码需要遵循 MLISA 语言规范，详见《Cambricon MLISA Developer Guide》。

7.5 cnrtcDestroyCode

```
cnrtcStatus cnrtcDestroyCode(cnrtcCode* code);
```

本函数用于销毁cnrtcCode 对象。

参数说明

- code[in]: cnrtcCode 对象。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_DESTROY_CODE_DOUBLE_FREE_ERROR
  cnrtcCode 对象已经销毁，本次调用属于重复销毁。

7.6 cnrtcCompileCode

```
cnrtcStatus cnrtcCompileCode(cnrtcCode code, int numOptions, const char** options);
```

本函数用于编译cnrtcCode 中的源代码。

参数说明

- code[in]: cnrtcCode 对象。
- numOptions[in]: 编译选项的数量，必须是非负整数。
- options[in]: 编译选项，字符串数组。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_COMPILE_CODE_NO_CODE_ERROR
  cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_COMPILE_CODE_PARAM_ERROR
  编译选项参数错误。

- CNRTC_COMPILE_CODE_WRITE_PERMISSION_ERROR
  不能获取系统的写权限。
- CNRTC_COMPILE_CODE_READ_PERMISSION_ERROR
  不能获取系统的读权限。
- CNRTC_COMPILE_CODE_ENVIRONMENT_ERROR
  运行环境异常。
- CNRTC_COMPILE_CODE_ERROR
  遇到源代码编译错误，请检查源代码。

注意事项

- 本函数需要在cnrtcCreateCode 之后调用。
- 若源代码是 Cambricon BANG C 代码，则需要满足以下约束：
  - 当前支持 CNCC 除 `-o` 外的全部编译选项，比如 `-O3` , `--bang-mlu-arch=...` 等，详情可参考《Cambricon BANG C Developer Guide》或 `cncc --help` 命令。
  - `--bang-mlu-arch=...` 编译选项用于指定代码的运行平台。为缩短编译时长，建议只添加一个平台。
  - 若在本函数后调用cnrtcGetFatBinary 或cnrtcGetFatBinarySize 函数，则必须添加 `--bang-fatbin-only` 编译选项。
- 若源代码是 MLISA 代码，则需要满足以下约束：
  - 当前支持 CNAS 除 `-i` , `--input` , `-o` , `--output` 外的全部编译选项，比如 `-O3` , `-a` 等，详情可参考《Cambricon MLISA Developer Guide》或 `cnas -h` 命令。
  - `--mlu-arch` 和 `-a` 等带有赋值的编译选项，需要把选项和赋值合并写为一个字符串，且在两者之间加一个空格，比如 `"-a mtp_270"` 作为一个字符串，而不能写成 `"-a"` 和 `"mtp_270"` 两个字符串。
  - `--mlu-arch` 和 `-a` 编译选项用于指定代码的运行平台。为缩短编译时长，建议只添加一个平台。
  - 若编译选项包含多个运行平台，即有多个 `--mlu-arch` 或 `-a` ，则必须再增加 `--fatbin` 编译选项，否则只编译最后一个平台。
  - 若在本函数后调用cnrtcGetFatBinary 或cnrtcGetFatBinarySize 函数，则必须添加 `--fatbin` 编译选项。

7.7 cnrtcGetFatBinary

```
cnrtcStatus cnrtcGetFatBinary(cnrtcCode code, void* bin);
```

本函数用于获取二进制输出 cnFatBinary。

参数说明

- code[in]: cnrtcCode 对象。
- bin[out]: 输出二进制的内存空间。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_GET_BIN_NO_CODE_ERROR
  cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_GET_BIN_NO_SPACE_ERROR
  接收二进制的内存指针未分配空间。
- CNRTC_NO_FATBIN_OPTION_ERROR
  在本函数调用前的cnrtcCompileCode 函数中，没有添加 `--bang-fatbin-only` 或 `--fatbin` 编译选项。

注意事项

- 本函数需要在cnrtcCompileCode 和cnrtcGetFatBinarySize 之后调用。
- 在本函数调用前的cnrtcCompileCode 函数中，若源代码是 Cambricon BANG C 代码，则必须添加 `--bang-fatbin-only` 编译选项，若源代码是 MLISA 代码，则必须添加 `--fatbin` 编译选项。

7.8 cnrtcGetFatBinarySize

```
cnrtcStatus cnrtcGetFatBinarySize(cnrtcCode code, unsigned int* binSize);
```


本函数用于获取二进制输出 cnFatBinary 所需的内存空间大小。

参数说明

- code[in]: cnrtcCode 对象。
- binSize[out]: 输出二进制的内存空间大小。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_GET_BIN_SIZE_NO_CODE_ERROR
  cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_GET_BIN_SIZE_NO_SPACE_ERROR
  接收二进制内存空间大小的内存指针未分配空间。
- CNRTC_NO_FATBIN_OPTION_ERROR
  在本函数调用前的cnrtcCompileCode 函数中，没有添加 `--bang-fatbin-only` 或 `--fatbin` 编译选项。

注意事项

- 本函数需要在cnrtcCompileCode 之后调用。
- 在本函数调用前的cnrtcCompileCode 函数中，若源代码是 Cambricon BANG C 代码，则必须添加 `--bang-fatbin-only` 编译选项，若源代码是 MLISA 代码，则必须添加 `--fatbin` 编译选项。

7.9 cnrtcGetCompilationOutput

```
cnrtcStatus cnrtcGetCompilationOutput(cnrtcCode code, void* output);
```

本函数用于获取编译输出。

参数说明

- code[in]: cnrtcCode 对象。
- output[out]: 编译输出的内存空间。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_NO_CODE_ERROR
  cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_NO_OUTPUT_SPACE_ERROR
  接收编译输出的内存指针未分配空间。

注意事项

- 本函数需要在cnrtcCompileCode 和cnrteGetCompilationOutputSize 之后调用。

7.10 cnrtcGetCompilationOutputSize

```
cnrtcStatus cnrtcGetCompilationOutputSize(cnrtcCode code, unsigned int* outputSize);
```

本函数用于获取编译输出所需的内存空间大小。

参数说明

- code[in]: cnrtcCode 对象。
- outputSize[out]: 编译输出占用的内存空间大小。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_NO_CODE_ERROR
  cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_NO_OUTPUT_SPACE_ERROR
  接收编译输出内存空间大小的内存指针未分配空间。

注意事项

- 本函数需要在cnrtcCompileCode 之后调用。

7.11 cnrtcGetCompilationLog

```
cnrtcStatus cnrtcGetCompilationLog(cnrtcCode code, char* log);
```

本函数用于获取编译错误记录。

参数说明

- code[in]: cnrtcCode 对象。
- log[out]: 输出错误记录的内存空间。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_GET_LOG_NO_CODE_ERROR
  cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。
- CNRTC_GET_LOG_NO_SPACE_ERROR
  接收错误记录的内存指针未分配空间。

注意事项

- 本函数需要在cnrtcCompileCode 和cnrtcGetCompilationLogSize 之后调用。
- 仅在cnrtcCompileCode 返回值 CNRTC_COMPILE_CODE_ERROR 时，才会产生错误记录。

7.12 cnrtcGetCompilationLogSize

```
cnrtcStatus cnrtcGetCompilationLogSize(cnrtcCode code, unsigned int* logSize);
```

本函数用于获取编译错误记录所需的内存空间大小。

参数说明

- code[in]: cnrtcCode 对象。
- logSize[out]: 编译错误记录所需的内存空间大小。

返回值描述

- CNRTC_SUCCESS
  函数执行成功。
- CNRTC_GET_LOG_SIZE_NO_CODE_ERROR
  cnrtcCode 对象未创建，需要提前调用cnrtcCreateCode 函数。

- CNRTC_GET_LOG_SIZE_NO_SPACE_ERROR
  接收编译错误记录内存空间大小的内存指针未分配空间。

注意事项

- 本函数需要在cnrtcCompileCode 之后调用。

8 示例代码

8.1 Cambricon BANG C 编译示例代码

本节介绍了使用 Cambricon CNRTC 编译 Cambricon BANG C，并用 CNDrv 运行的示例代码，最后给出编译、运行脚本。

```
// ***************************************************************************
// Copyright (C) [2021-2025] by Cambricon, Inc.
// ***************************************************************************
#include <iostream>  // std::cout,cin
#include <cstdlib>   // atoi
#include <cmath>     // sqrtf, fabsf
#include "cnrtc.h"
#include "cn_api.h"

#define CHECK_CNRTC_SUCCESS(func_return) {                          \
  if (func_return != CNRTC_SUCCESS) {                               \
    std::cerr << "Error line " << __LINE__ << " : "                 \
              << cnrtcTransStatusToString(func_return) << "\n";     \
    return false;                                                   \
  }                                                                 \
}

#define CHECK_DRV_SUCCESS(func_return) {                            \
  if (func_return != CN_SUCCESS) {                                  \
    const char* str;                                                \
    cnGetErrorName(func_return, &str);                              \
    std::cerr << "Error line " << __LINE__ << " : " << str << "\n"; \
    return false;                                                   \
  }                                                                 \
}

// \brief   checkPass compares the relative difference between two arrays
//          and prints PASSED if the difference is less than diffThres
```

```
//          or prints FAILED if the difference is greater than diffThres.
// \param   [in] inX The array of input data.
// \param   [in] mluY The array of MLU output data.
// \param   [in] cpuY The array of CPU output data.
// \param   [in] len Array length.
// \param   [in] diffThres Threshold.
// \return  bool: true if the difference is less than diffThres or
//                false if the difference is greater than diffThres.
bool checkPass(float *inX, float *mluY, float *cpuY, int len, float diffThres) {
  bool res = true;
  for (int i = 0; i < len; ++i) {
    if (fabsf(mluY[i] - cpuY[i]) / cpuY[i] > diffThres) {
      std::cout << "No." << i << " is wrong! input = " << inX[i]
                << ", mlu = " << mluY[i] << ", cpu = " << cpuY[i] << "\n";
      res = false;
    }
  }
  if (res) {
    std::cout << "PASSED ! ^_^\n";
  } else {
    std::cout << "FAILED ! >_<\n";
  }
  return res;
}

// \brief   basicCompileAndRun Cambricon CNRTC basic usage example.
// \param   [in] srcCode The Cambricon BANG C source code.
// \param   [in] dataSize The data size of input and output data.
// \param   [in] pHostX The host pointer of input data.
// \param   [out] pHostY The host pointer of output data.
// \return  bool: If the compilation and running is successful, it returns true,
//                otherwise, it returns false.
bool basicCompileAndRun(const char* srcCode, int dataSize, void* pHostX, void* pHostY) {
  // 1. Create and compile code.
  cnrtcCode code = NULL;
  CHECK_CNRTC_SUCCESS(cnrtcCreateCode(&code, srcCode, NULL, 0, NULL, NULL));
  const char *opts[] = {"-O3", "--bang-fatbin-only", "--bang-mlu-arch=mtp_220",
                        "--bang-mlu-arch=mtp_270", "--bang-mlu-arch=mtp_290",
                        "--bang-mlu-arch=tp_322", "--bang-mlu-arch=mtp_372",
                        "--bang-mlu-arch=mtp_592"};
  cnrtcStatus res = cnrtcCompileCode(code, 8, opts);
```

```
// 2. Check compilation results and print error log.
if (res != CNRTC_SUCCESS) {
  std::cerr << "cnrtcCompileCode failed : " << cnrtcTransStatusToString(res) << "\n";

  unsigned int logSize = 0;
  CHECK_CNRTC_SUCCESS(cnrtcGetCompilationLogSize(code, &logSize));
  char* log = (char*)malloc(logSize);
  CHECK_CNRTC_SUCCESS(cnrtcGetCompilationLog(code, log));

  std::cout << "log:\n" << log << "\n";
  free(log);
  CHECK_CNRTC_SUCCESS(cnrtcDestroyCode(&code));
  return false;
}

// 3. Get FatBinary, which is the compilation output.
unsigned int binSize = 0;
CHECK_CNRTC_SUCCESS(cnrtcGetFatBinarySize(code, &binSize));
void* bin = malloc(binSize);
CHECK_CNRTC_SUCCESS(cnrtcGetFatBinary(code, bin));
```



```
// 4. Destroy code.
CHECK_CNRTC_SUCCESS(cnrtcDestroyCode(&code));

// 5. Open MLU device and create context.
CHECK_DRV_SUCCESS(cnInit(0));
int device_count = 0;
CHECK_DRV_SUCCESS(cnDeviceGetCount(&device_count));
if (device_count < 1) {
  std::cerr << "No MLU device !\n";
  free(bin);
  return false;
}
CNdev dev;
CHECK_DRV_SUCCESS(cnDeviceGet(&dev, 0));
CNcontext context;
CHECK_DRV_SUCCESS(cnCtxCreate(&context, 0, dev));

// 6. Create module using FatBinary and get kernel in module.
CNmodule module;
CHECK_DRV_SUCCESS(cnModuleLoadFatBinary(bin, &module));
free(bin);
```



```
  CNkernel kernel;
  CHECK_DRV_SUCCESS(cnModuleGetKernel(module, "devKernel", &kernel));

  // 7. Prepare MLU data.
  CNaddr pDevX, pDevY;
  CHECK_DRV_SUCCESS(cnMalloc(&pDevX, dataSize));
  CHECK_DRV_SUCCESS(cnMalloc(&pDevY, dataSize));
  CHECK_DRV_SUCCESS(cnMemcpy(pDevX, (CNaddr)pHostX, dataSize));

  // 8. Invoke kernel.
  void *params[] = {&pDevX, &pDevY};
  CNqueue queue;
  CHECK_DRV_SUCCESS(cnCreateQueue(&queue, 0));
  CHECK_DRV_SUCCESS(cnInvokeKernel(kernel, 1, 1, 1, CN_KERNEL_CLASS_BLOCK, 0, queue, params,␣
↪NULL));
  CHECK_DRV_SUCCESS(cnQueueSync(queue));
  CHECK_DRV_SUCCESS(cnDestroyQueue(queue));

  // 9. Get MLU output data.
  CHECK_DRV_SUCCESS(cnMemcpy((CNaddr)pHostY, pDevY, dataSize));
```



```
  // 10. Free resource.
  CHECK_DRV_SUCCESS(cnFree(pDevX));
  CHECK_DRV_SUCCESS(cnFree(pDevY));
  CHECK_DRV_SUCCESS(cnModuleUnload(module));
  CHECK_DRV_SUCCESS(cnCtxDestroy(context));
  return true;
}

int main(int argc, char **argv) {
  // 1. Generate source code.
  const char* srcCode = "\
    #include \"bang.h\"                                          \n\
    #define LEN 1024                                            \n\
    extern \"C\" __mlu_global__ void devKernel(void *x, void *y) { \n\
      __nram__ float nx[LEN];                                   \n\
      __nram__ float ny[LEN];                                   \n\
      __memcpy(nx, x, LEN * sizeof(float), GDRAM2NRAM);         \n\
      __bang_active_rsqrt(ny, nx, LEN);                         \n\
      __memcpy(y, ny, LEN * sizeof(float), NRAM2GDRAM);         \n\
    }                                                           \n";
```

```
  // 2. Prepare CPU data.
  int dataNum = 1024;
  int dataSize = dataNum * sizeof(float);
  float *pHostX = (float *)malloc(dataSize);
  float *pHostY = (float *)malloc(dataSize);
  float *cpuY = (float *)malloc(dataSize);

  for (int j = 0; j < dataNum; ++j) {
    pHostX[j] = (j + 1) / 100.0;  // 0.01 ~ 10.24
    cpuY[j] = 1.0 / sqrtf(pHostX[j]);
  }

  // 3. Compile and run on MLU.
  bool res = basicCompileAndRun(srcCode, dataSize, pHostX, pHostY);
  if (res) {
    res &= checkPass(pHostX, pHostY, cpuY, dataNum, 0.004);
  }

  // 4. Free resource.
  free(pHostX);
  free(pHostY);
  free(cpuY);
  return res ? 0 : 1;
}
```


将上面代码保存为 main.cpp 文件，准备好寒武纪基础软件平台的环境，终端执行下面命令编译：

```
$ export NEUWARE_HOME="/path/to/neuware"  # 默认安装后的配置为 export NEUWARE_HOME="/usr/local/
↪neuware"
$ g++ main.cpp -o main.out -I ${NEUWARE_HOME}/include -L ${NEUWARE_HOME}/lib64 -lcnrtc -lcndrv
```

在装有 MLU 设备的机器上设置环境变量和运行：

```
$ export NEUWARE_HOME="/path/to/neuware"  # 默认安装后的配置为 export NEUWARE_HOME="/usr/local/
↪neuware"
$ export PATH="${NEUWARE_HOME}/bin":$PATH
$ export LD_LIBRARY_PATH="${NEUWARE_HOME}/lib64":$LD_LIBRARY_PATH
$ ./main.out
```

若打印输出 PASSED ! ^_^ 即为运行成功。

8.2 MLISA 编译示例代码

本节介绍了使用 Cambricon CNRTC 编译 MLISA，并用 CNDrv 运行的示例代码，最后给出编译、运行脚本。

```
// ************************************************************************
// Copyright (C) [2021-2025] by Cambricon, Inc.
// ************************************************************************
#include <iostream>  // std::cout,cin
#include <cstdlib>   // atoi
#include <cmath>     // sqrtf, fabsf
#include "cnrtc.h"
#include "cn_api.h"

#define CHECK_CNRTC_SUCCESS(func_return) {                          \
  if (func_return != CNRTC_SUCCESS) {                               \
    std::cerr << "Error line " << __LINE__ << " : "                 \
              << cnrtcTransStatusToString(func_return) << "\n";     \
    return false;                                                   \
  }                                                                 \
}


#define CHECK_DRV_SUCCESS(func_return) {                            \
  if (func_return != CN_SUCCESS) {                                  \
    const char* str;                                                \
    cnGetErrorName(func_return, &str);                              \
    std::cerr << "Error line " << __LINE__ << " : " << str << "\n"; \
    return false;                                                   \
  }                                                                 \
}

// \brief   checkPass compares the relative difference between two arrays
//          and prints PASSED if the difference is less than diffThres
//          or prints FAILED if the difference is greater than diffThres.
// \param   [in] inX The array of input data.
// \param   [in] mluY The array of MLU output data.
// \param   [in] cpuY The array of CPU output data.
// \param   [in] len Array length.
// \param   [in] diffThres Threshold.
// \return  bool: true if the difference is less than diffThres or
//                false if the difference is greater than diffThres.
bool checkPass(float *inX, float *mluY, float *cpuY, int len, float diffThres) {
```

```
  bool res = true;
  for (int i = 0; i < len; ++i) {
    if (fabsf(mluY[i] - cpuY[i]) / cpuY[i] > diffThres) {
      std::cout << "No." << i << " is wrong! input = " << inX[i]
                << ", mlu = " << mluY[i] << ", cpu = " << cpuY[i] << "\n";
      res = false;
    }
  }
  if (res) {
    std::cout << "PASSED ! ^_^\n";
  } else {
    std::cout << "FAILED ! >_<\n";
  }
  return res;
}

// \brief   MLISACompileAndRun Cambricon CNRTC compile MLISA example.
// \param   [in] srcCode The MLISA source code.
// \param   [in] dataSize The data size of input and output data.
// \param   [in] pHostX The host pointer of input data.
// \param   [out] pHostY The host pointer of output data.
// \return  bool: If the compilation and running is successful, it returns true,
//                otherwise, it returns false.
bool MLISACompileAndRun(const char* srcCode, int dataSize, void* pHostX, void* pHostY) {
  // 1. Create and compile code.
  cnrtcCode code = NULL;
  CHECK_CNRTC_SUCCESS(cnrtcCreateCodeV2(&code, srcCode, CNRTC_CODE_MLISA, NULL, 0, NULL,␣
↪NULL));
  const char *opts[] = {"-O3", "--fatbin", "-a mtp_220", "-a mtp_270", "-a mtp_290",
                        "-a tp_322", "-a mtp_372", "-a mtp_592"};
  cnrtcStatus res = cnrtcCompileCode(code, 8, opts);

  // 2. Check compilation results and print error log.
  if (res != CNRTC_SUCCESS) {
    std::cerr << "cnrtcCompileCode failed : " << cnrtcTransStatusToString(res) << "\n";

    unsigned int logSize = 0;
    CHECK_CNRTC_SUCCESS(cnrtcGetCompilationLogSize(code, &logSize));
    char* log = (char*)malloc(logSize);
    CHECK_CNRTC_SUCCESS(cnrtcGetCompilationLog(code, log));

    std::cout << "log:\n" << log << "\n";
```

```
  free(log);
  CHECK_CNRTC_SUCCESS(cnrtcDestroyCode(&code));
  return false;
}

// 3. Get FatBinary, which is the compilation output.
unsigned int binSize = 0;
CHECK_CNRTC_SUCCESS(cnrtcGetFatBinarySize(code, &binSize));
void* bin = malloc(binSize);
CHECK_CNRTC_SUCCESS(cnrtcGetFatBinary(code, bin));

// 4. Destroy code.
CHECK_CNRTC_SUCCESS(cnrtcDestroyCode(&code));

// 5. Open MLU device and create context.
CNresult res2 = cnInit(0);
if (res2 != CN_SUCCESS) {
  const char* str;
  cnGetErrorName(res2, &str);
  std::cerr << "cnInit() error : " << str << "\n";
  free(bin);
  return false;
}
int device_count = 0;
CHECK_DRV_SUCCESS(cnDeviceGetCount(&device_count));
CNdev dev;
CHECK_DRV_SUCCESS(cnDeviceGet(&dev, 0));
CNcontext context;
CHECK_DRV_SUCCESS(cnCtxCreate(&context, 0, dev));

// 6. Create module using FatBinary and get kernel in module.
CNmodule module;
CHECK_DRV_SUCCESS(cnModuleLoadFatBinary(bin, &module));
free(bin);
CNkernel kernel;
CHECK_DRV_SUCCESS(cnModuleGetKernel(module, "devKernel", &kernel));

// 7. Prepare MLU data.
CNaddr pDevX, pDevY;
CHECK_DRV_SUCCESS(cnMalloc(&pDevX, dataSize));
CHECK_DRV_SUCCESS(cnMalloc(&pDevY, dataSize));
CHECK_DRV_SUCCESS(cnMemcpy(pDevX, (CNaddr)pHostX, dataSize));
```

```
  // 8. Invoke kernel.
  void *params[] = {&pDevX, &pDevY};
  CNqueue queue;
  CHECK_DRV_SUCCESS(cnCreateQueue(&queue, 0));
  CHECK_DRV_SUCCESS(cnInvokeKernel(kernel, 1, 1, 1, CN_KERNEL_CLASS_BLOCK, 0, queue, params,␣
↪NULL));
  CHECK_DRV_SUCCESS(cnQueueSync(queue));
  CHECK_DRV_SUCCESS(cnDestroyQueue(queue));

  // 9. Get MLU output data.
  CHECK_DRV_SUCCESS(cnMemcpy((CNaddr)pHostY, pDevY, dataSize));

  // 10. Free resource.
  CHECK_DRV_SUCCESS(cnFree(pDevX));
  CHECK_DRV_SUCCESS(cnFree(pDevY));
  CHECK_DRV_SUCCESS(cnModuleUnload(module));
  CHECK_DRV_SUCCESS(cnCtxDestroy(context));
  return true;
}

int main(int argc, char **argv) {
  // 1. Generate source code.
  const char* srcCode = "\
      .mlisa 2.0                                \n\
      .arch MLU270                              \n\
      .stack nram                               \n\
      .nram .align 64 .b8 .len 4096 _nx[4096]; \n\
      .nram .align 64 .b8 .len 4096 _ny[4096]; \n\
                                                \n\
      .visible .kernel devKernel(               \n\
          .arg .ptr devKernel_arg_0,            \n\
          .arg .ptr devKernel_arg_1)            \n\
      {                                         \n\
        .gpr %r<2>;                             \n\
        ld.gpr.arg  %r0, devKernel_arg_0, 6;    \n\
        ld.gpr.arg  %r1, devKernel_arg_1, 6;    \n\
        ld.nram.gdram  _nx, [%r0], 4096;        \n\
        active.rsqrt.nram.f32  _ny, _nx, 1024; \n\
        st.gdram.nram  [%r1], _ny, 4096;        \n\
        exit;                                   \n\
      }                                         \n";
```

```
  // 2. Prepare CPU data.
  int dataNum = 1024;
  int dataSize = dataNum * sizeof(float);
  float *pHostX = (float *)malloc(dataSize);
  float *pHostY = (float *)malloc(dataSize);
  float *cpuY = (float *)malloc(dataSize);

  for (int j = 0; j < dataNum; ++j) {
    pHostX[j] = (j + 1) / 100.0;  // 0.01 ~ 10.24
    cpuY[j] = 1.0 / sqrtf(pHostX[j]);
  }

  // 3. Compile and run on MLU.
  bool res = MLISACompileAndRun(srcCode, dataSize, pHostX, pHostY);
  if (res) {
    res &= checkPass(pHostX, pHostY, cpuY, dataNum, 0.004);
  }

  // 4. Free resource.
  free(pHostX);
  free(pHostY);
  free(cpuY);
  return res ? 0 : 1;
}
```


将上面代码保存为 main.cpp 文件，准备好寒武纪基础软件平台的环境，终端执行下面命令编译：

```
$ export NEUWARE_HOME="/path/to/neuware"  # 默认安装后的配置为 export NEUWARE_HOME="/usr/local/
↪neuware"
$ g++ main.cpp -o main.out -I ${NEUWARE_HOME}/include -L ${NEUWARE_HOME}/lib64 -lcnrtc -lcndrv
```

在装有 MLU 设备的机器上设置环境变量和运行：

```
$ export NEUWARE_HOME="/path/to/neuware"  # 默认安装后的配置为 export NEUWARE_HOME="/usr/local/
↪neuware"
$ export PATH="${NEUWARE_HOME}/bin":$PATH
$ export LD_LIBRARY_PATH="${NEUWARE_HOME}/lib64":$LD_LIBRARY_PATH
$ ./main.out
```

若打印输出 PASSED ! ^_^ 即为运行成功。

注解：

更多示例详见 CNToolkit 包的 samples 目录：`neuware/samples/cnrtc/` 。

Cambricon 寒武纪

9 FAQ

1. Cambricon CNRTC 相比 CNCC/CNAS 有什么不同，优势是什么?

CNCC 是 Cambricon BANG C 语言的编译器，CNAS 是 MLISA 语言的汇编器，他们都是 AOT 编译，而 Cambricon CNRTC 支持 Cambricon BANG C 语言和 MLISA 语言的运行时编译，是 JIT 编译。Cambricon CNRTC 比 CNCC/CNAS 的编译速度更快。

2. Cambricon CNRTC 支持哪些编译选项?

Cambricon CNRTC 是为用户在其应用程序中编译 Cambricon BANG C 或 MLISA 代码而设计，现已支持除 `-o` 外的全部 CNCC 编译选项和除 `-i` , `--input` , `-o` , `--output` 外的全部 CNAS 编译选项。

`-i` 和 `-o` 不支持的原因是 Cambricon CNRTC 有自己的编译输入和输出获取接口cnrtcCreateCode , cnrtcCreateCodeV2 , cnrtcGetCompilationOutput 和cnrtcGetCompilationOutputSize ，无需用户使用 `-i` 和 `-o` 选项来指定输入输出文件名。关于编译选项请参见cnrtcCompileCode 接口说明。

3. 如何让 Cambricon CNRTC 编译得到 MLU 可执行的二进制? CNRT 能否执行?

用户需要在cnrtcCompileCode 函数中添加 `--bang-fatbin-only` (针对 Cambricon BANG C 语言) 或 `--fatbin` (针对 MLISA 语言) 编译选项进行编译，然后使用cnrtcGetFatBinary 和cnrtcGetFatBinarySize 接口获得编译结果 `cnFatBinary` 。当前仅有 CNDRV 可以执行 `cnFatBinary` 二进制，CNRT 不支持。详细用法可参考示例代码 。

4. 若想使用cnrtcGetFatBinary 或cnrtcGetFatBinarySize 接口，必须在cnrtcCompileCode 函数中添加 `--bang-fatbin-only` 或 `--fatbin` 编译选项吗?

是的。因为从 Cambricon CNRTC v0.4.0 版本起，已支持除 `-o` 外的全部 CNCC 编译选项，从 Cambricon CNRTC v0.5.0 版本起，已支持除 `-i` , `--input` , `-o` , `--output` 外的全部 CNAS 编译选项，用户可得到多种编译输出，而cnrtcGetFatBinary 和cnrtcGetFatBinarySize 接口只能输出 `cnFatBinary` ，因此需要用户在cnrtcCompileCode 函数中添加 `--bang-fatbin-only` (针对 Cambricon BANG C 语言) 或 `--fatbin` (针对 MLISA 语言) 编译选项来指明获得 `cnFatBinary` 。

如需获得其他编译输出，则不能使用cnrtcGetFatBinary 和cnrtcGetFatBinarySize ，需要使用cnrtcGetCompilationOutput 和cnrtcGetCompilationOutputSize 。

