寒武纪运行时库用户手册

版本 *4.3.0*

2020 年 05 月 06 日

Cambricon
寒 武 纪

目录

目录 i

插图目录 1

表格目录 2

1 版权声明 3

2 前言 5
- 2.1 版本记录 5
- 2.2 更新历史 5

3 快速入门 8
- 3.1 安装 CNRT 8
- 3.2 使用 CNRT 运行离线模型 8
- 3.3 CNRT 开发样例 9
  - 3.3.1 开发样例目录结构 9
  - 3.3.2 运行开发样例 9

4 介绍 11
- 4.1 概述 11
- 4.2 基本概念 11
  - 4.2.1 Function 12
    - 4.2.1.1 示例代码 12
  - 4.2.2 Context 13
    - 4.2.2.1 相关接口介绍 14
    - 4.2.2.2 共享 Context 权值和指令内存 15
    - 4.2.2.3 离线模式示例 15
- 4.3 功能 17
  - 4.3.1 功能介绍 17
  - 4.3.2 设备管理 17
    - 4.3.2.1 设备初始化 17
    - 4.3.2.2 设备操作 17
    - 4.3.2.3 设备销毁 19

4.3.2.4 示例 19
4.3.3 内存管理 20
4.3.3.1 主机端内存管理 20
4.3.3.2 MLU 端内存管理 21
4.3.3.3 MLU 与主机间的内存拷贝 22
4.3.4 队列 23
4.3.4.1 队列的属性 23
4.3.4.2 接口介绍 24
4.3.4.3 示例 25
4.3.5 同步机制 25
4.3.5.1 同步机制执行流程 25
4.3.6 任务执行 25
4.3.6.1 任务执行流程 25
4.3.6.2 示例 26
4.3.7 Notifier 26
4.3.7.1 Notifier 机制 26
4.3.7.2 Notifier 计时步骤 27
4.3.7.3 示例 27
4.3.8 离线模型管理 28
4.3.8.1 功能介绍 28
4.3.8.2 模型加载流程 29
4.3.8.3 模型版本兼容性 29
4.3.8.4 示例 30

5 编程模型 **31**
5.1 MLU220 编程模型 31
5.1.1 MLU220 数据读写优化 31
5.1.1.1 CPU 写数据到内存 31
5.1.1.2 MLU 写入数据到内存 31
5.2 指定 DDR 通道和 Cluster 来执行 Cluster 任务 32
5.2.1 示例代码 32
5.3 BANG C 混合编程 32
5.3.1 功能接口介绍 32
5.3.1.1 参数操作 32
5.3.1.2 数据占位 33
5.3.1.3 内核调用 33
5.3.2 BANG C 混合编程示例 36

6 调试方法 **38**
6.1 使用 loginfo 记录 API 日志 38

7 示例程序 40
7.1 离线模型示例程序 40
7.2 异步拷贝示例程序 43
7.3 任务执行示例程序 47
7.4 指定 DDR 通道和 Cluster 来执行 Cluster 任务 49
7.5 共享 Context 权值和指令内存示例 53

8 性能调优指南 59
8.1 通过设置 DDR 通道和 MLU 亲和性提高访存效率 59
8.2 内核调用编程模型的优化 59
8.3 提升 MLU220 数据读写性能 59

9 CNRT 环境变量 61
9.1 CNRT_PRINT_INFO 61
9.2 CNRT_GET_HARDWARE_TIME 61
9.3 CNRT_DUMP_MLUGPR 61
9.4 CNRT_DUMP_MLURAM 62
9.5 CNRT_DUMP_MLUSCALAR 62
9.6 CNRT_DUMP_MLUVECTOR 62
9.7 CNRT_BANG_PRINTF_LIMIT 62
9.8 CNRT_DEFAULT_DEVICE 63
9.9 CNRT_BANG_PRINTF_NOT_ENABLE 63

插图目录

4.1 寒武纪软件栈 11
4.2 单个队列任务执行机制 23
4.3 多个队列任务执行机制 24
4.4 Notifier 执行方法 27

表格目录

2.1 版本记录 5

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美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。

本指南为版权所有并受全世界版权法律和条约条款的保护。未经寒武纪的事先书面许可，不可以任何方

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
- © 2020 中科寒武纪科技股份有限公司保留一切权利。

Cambricon
寒 武 纪

2 前言

2.1 版本记录

表 2.1: 版本记录

文档名称	寒武纪运行时库用户手册
版本号	V4.3.0
作者	Cambricon
修改日期	2020.04.16

2.2 更新历史

- **V4.3.0**

  更新时间：2020 年 04 月 16 号

  更新内容：

  – Context 支持共享 Context 权值和指令内存。

  – MLU220 EDGE 支持仅对某一块区域做内存地址映射和某一段数据做同步。详情查看MLU220 编程模型。

  – 增加释放指定内存空间的相关描述：释放内存空间时，用户无需指定设备。

  – 新增快速入门 章节，介绍如何安装、快速使用 CNRT 运行离线模型以及如何运行寒武纪提供的 CNRT 样例。

  – 对已支持功能的内容优化及结构调整。

- **V4.2.1**

  更新时间：2020 年 2 月 24 号

  更新内容：

  – 新增环境变量CNRT_BANG_PRINTF_NOT_ENABLE 介绍。

  – 新增性能调优指南。

  – 新增更改设备设置介绍。支持初始化设备时，系统自动设置默认设备。详情查看设备操作。

Copyright © 2020 Cambricon Corporation.

- **V4.2.0**

  更新时间：2019 年 12 月 31 号

  更新内容：

  – 新增 BANG C 混合编程 kenel 调用 API V3 版本介绍。

    详情请查看内核调用 中“内核调用接口 V3 版”小节。

- **V4.1.1**

  更新时间：2019 年 12 月 15 号

  更新内容：

  – 无。

- **V4.1.0**

  更新时间：2019 年 11 月 30 号

  更新内容：

  – 支持 MLU220。

  – 新增 `CNRT_DEFAULT_DEVICE` 环境变量。

  – 异步拷贝功能介绍。

  – MLU220 编程模型介绍。

  – 指定 DDR 通道和 Cluster 执行 Cluster 任务。

  – 更新示例。

- **V4.0.1**

  更新时间：2019 年 11 月 15 号

  更新内容：

  – 无。

- **V4.0.0**

  更新时间：2019 年 10 月 23 号

  更新内容：

  – 由于功能变更，删除“内存通道设置”、“默认队列”、“指令数据共享”章节。

  – 更新下面章节中的功能描述，包括更新为新版 API 函数以及删掉不再支持的 API 函数等：

    * 任务队列及同步机制
    * 离线模型
    * 任务执行
    * Function
    * 内存管理
    * Context

  – 更新手册中所有示例。

  – 对已支持功能的内容优化及结构调整。

- **V3.14.0**

  更新时间：2019 年 9 月 23 号

  更新内容：

  – 无

- **V3.13.0**

  更新时间：2019 年 8 月 15 号

  更新内容：

  - 初始化版本。

Cambricon
寒 武 纪

3 快速入门

本章介绍了如何安装和使用 CNRT 以及如何运行寒武纪提供的 CNRT 样例。

3.1 安装 CNRT

CNRT 的安装不依赖于任何第三方库和环境。用户只需安装 Neuware SDK 软件包即可完成 CNRT 的安装。安装详细步骤，请参考《寒武纪 Neuware SDK 软件包安装升级使用手册》。

3.2 使用 CNRT 运行离线模型

本节介绍了如何基于 CNRT 运行一个离线模型。运行前，用户需要准备好离线模型文件。了解如何生成离线模型，请参考《寒武纪 CNML 用户手册》中“离线运行模型”章节。

执行下面步骤运行离线模型：

1. 调用 `cnrtInit()` API，初始化设备。
2. 调用 `cnrtLoadModel()` API，加载离线模型。
3. 调用 `cnrtSetCurrentDevice()` API，设置使用的 MLU 设备。
4. 调用 `cnrtExtractFunction ()` API，提取离线模型中的模型信息。
5. 通过 `cnrtGetInputDataSize()` 和 `cnrtGetOutputDataSize()` API 获得输入和输出数据的内存大小。
6. 调用 `cnrtMalloc()` API，为 MLU 输入数据分配内存指定空间。
7. 调用 `cnrtMemcpy()` API，同步拷贝主机端数据到 MLU 端。
8. 调用 `cnrtMalloc()` API，为 MLU 输出数据分配内存指定空间。
9. 设置 Context。
   (a) 调用 `cnrtCreateRuntimeContext()` API，创建 Context。
   (b) 调用 `cnrtSetRuntimeContextDeviceId()` API，绑定设备。
   (c) 调用 `cnrtInitRuntimeContext()` API，初始化 Context。
   (d) 调用 `cnrtRuntimeContextCreateQueue()` API，创建队列。
10. 调用 `cnrtInvokeRuntimeContext()` API，将任务下发到队列。
11. 调用 `cnrtSyncQueue()` API，同步任务。
12. 调用 `cnrtMemcpy()` API，将计算结果从 MLU 拷出到 CPU。

详细代码，请参考离线模型示例程序。

Copyright © 2020 Cambricon Corporation.

3.3 CNRT 开发样例

寒武纪 CNRT 开发样例为用户提供了离线模型、运行脚本以及开发样例代码，帮助用户快速体验如何使用 CNRT 运行离线模型。用户可以直接通过脚本运行样例代码，无需修改任何配置。

3.3.1 开发样例目录结构

开发样例存放于 `Cambricon-MLU270.tar.gz` 文件中 `cnrt` 文件夹下。开发样例包含的文件如下：

- script 文件夹：编译和运行开发样例的脚本文件。
  - compile_cnrt.sh：编译 CNRT 开发样例执行的脚本文件。
  - runExampleTests.sh：运行 CNRT 开发样例时执行的脚本文件。
- tests 文件夹：CNRT 开发样例。
  - cache_model：离线模型文件。mlp_mlu220.mef 用于 MLU220 平台，mlp_mlu270.mef 用于 MLU270 平台。
  - all_test.c：main 函数执行入口。用于运行开发样例。
  - offline.c：开发样例，完成一个离线模型任务。
  - parallel.c：开发样例，并行完成多个离线模型任务。
  - serial.c：开发样例，串行完成多个离线模型任务。
  - tests.h：开发样例函数的声明。
- CMakeLists.txt：CMake 文件，编译开发样例时使用。用户无需做任何设置和改动。
- README：开发样例说明和使用指南。

3.3.2 运行开发样例

运行开发样例的步骤如下：

1. 执行下面命令设置动态链接 CNRT 库。将命令中 `Cambricon NEUWARE PATH` 替换为寒武纪软件栈的动态链接库路径，即存放 libcnrt.so 和 libcndrv.so 的路径。

   ```
   export NEUWARE_HOME="Cambricon NEUWARE PATH"
   ```

2. 编译样例。在 `script` 目录下运行下面命令：

   ```
   ./compile_cnrt.sh
   ```

3. 运行样例。在 `script` 目录下运行下面命令：

   ```
   ./runExampleTests.sh
   ```

   运行后，返回下面界面：

```
please choose which test to run:
********************************
1.  offline_test ---- run an offline model on mlu
2.  serial_test ---- run offline models serially on mlu
3.  parallel_test ---- run offline models in parallel on mlu
********************************
```

4. 在返回界面输入想要运行的样例编号。运行后，返回下面界面：

```
please choose which test to run:
********************************
1. MLU270
2. MLU220
********************************
```

5. 在返回界面输入想要运行的平台编号。

CNRT 样例将会在用户选定的平台上运行。

Copyright © 2020 Cambricon Corporation.

4 介绍

CNRT（Cambricon Neuware Runtime Library，寒武纪运行时库）提供了一套面向 MLU（Machine Learning Unit，寒武纪机器学习单元）设备的高级别的接口，用于主机与 MLU 设备之间的交互。CNRT 作为寒武纪软件系统最底层支撑，所有其他的寒武纪软件运行都需要调用 CNRT 接口。

该用户手册涵盖了寒武纪 MLU270 和 MLU220 的功能介绍。

4.1 概述

本节介绍了寒武纪软件栈，结构如下图所示。

软件栈最上层的机器学习应用可以直接采用各种编程框架的编程接口，如 TensorFlow、Caffe、MXNet 等，间接通过 CNML（Cambricon Neuware Machine Learning Library，寒武纪机器学习库）调用 CNRT 进行软件编程。也可以直接调用 CNRT，运行上述过程所生成的离线模型，减少软件架构的中间开销，提高实际运行效率。

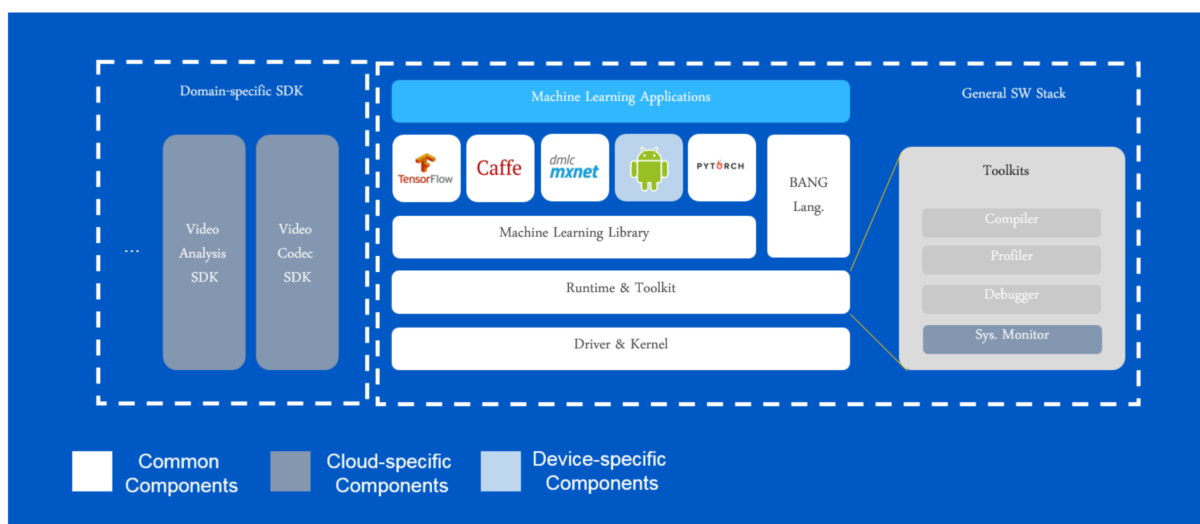

图 4.1: 寒武纪软件栈

4.2 基本概念

本章描述 CNRT 编程中所涉及的具体概念。

Copyright © 2020 Cambricon Corporation.

4.2.1 Function

Function 是 CNRT 的数据对象，用来描述运算单元信息。除此之外，Function 还包括若干离线计算的数据地址、数据偏移以及一些硬件计算属性。硬件计算属性主要有栈基址、静态数据基址、计算类型、设备数量、设备状态等。

CNRT 提供了 Function 操作的相关接口，用户可以从离线模型中解析出 Function 并获取 Function 的相关信息。了解如何创建离线模型，查看离线模型管理。

Function 的使用流程如下：

1. 创建 cnrtModel_t 结构体。
2. 调用 cnrtLoadModel() API 将离线模型文件中的数据写入到创建的 cnrtModel_t 结构体变量中。
3. 调用 cnrtGetFunctionNumber() API 获取模型中包含的 Function 数量。
4. 调用 cnrtCreateFunction() API 初始化 cnrtFunction_t 结构体变量。
5. 调用 cnrtExtractFunction() API 提取离线模型中的模型信息，并将其写入上一步创建的 cnrtFunction_t 结构体变量中。
6. 运算完毕后通过 cnrtDestroyFunction() API 销毁 Function。

4.2.1.1 示例代码

```
    cnrtInit(0);
    unsigned dev_num;
    cnrtGetDeviceCount(&dev_num);
    if (dev_num == 0)
    return NULL;
    cnrtDev_t dev;
    cnrtGetDeviceHandle(&dev, 0);
    cnrtSetCurrentDevice(dev);

    // load model and get function
    cnrtModel_t model;
    char fname[100] = "";
    // The name parameter represents the name of the offline model file. It is also the name␣
↪of a function in the offline model file.
    strcat(fname, (char *)name);
    strcat(fname, ".mef");
    printf("load file: %s\n", fname);
    cnrtLoadModel(&model, fname);

    int64_t totalMem;
    cnrtGetModelMemUsed(model, &totalMem);
```

Copyright © 2020 Cambricon Corporation.

```
    printf("total memory used: %" PRId64 " Bytes\n", totalMem);

    int func_num = 0;
    cnrtGetFunctionNumber(model, &func_num);
    printf("model function number is %d\n", func_num);

    // setup function
    cnrtFunction_t function;
    cnrtCreateFunction(&function);
    cnrtExtractFunction(&function, model, (char *)name);

    int64_t mem;
    cnrtGetFunctionMemUsed(function, &mem);
    printf("function total memory used: %" PRId64 " Bytes\n", mem);

    uint64_t local_mem_size;
    cnrtQueryModelLocalMemSize(model, &local_mem_size);
    printf("model kernels' biggest local size is: %" PRId64 " MB\n", local_mem_size);

...

    // free resource
    cnrtDestroyFunction(function);
    cnrtUnloadModel(model);
    ...
```

4.2.2 Context

Context 是 Function 的上下文描述，用于将计算资源和 Function 本身解除绑定，实现同一个 Function 可以根据不同的配置，实例化到不同的硬件设备。MLU 为用户提供以下 Context 接口。更多 API 详情，请查看《寒武纪 CNRT 开发者手册》。

使用 Context 的一般流程如下：

1. 调用 `cnrtCreateRuntimeContext()` 接口，创建 Context。
2. 调用 `cnrtSetRuntimeContextDeviceId()` 接口，设置 Context 设备 ID。
3. 调用 `cnrtInitRuntimeContext()` 接口，初始化 Context。
4. 调用 `cnrtRuntimeContextCreateQueue()` 接口，创建 Context 队列。
5. 调用 `cnrtInvokeRuntimeContext()` 接口，调用 Context。
6. 调用 `cnrtDestoryRuntimeContext()` 接口，销毁 Context。

4.2.2.1 相关接口介绍

创建 Context

```
cnrtCreateRuntimeContext(cnrtRuntimeContext_t *pctx, cnrtFunction_t function, void␣
↪*extra)
```

设置 Context 设备 ID

如果已通过CNRT_DEFAULT_DEVICE 环境变量指定了使用设备，下面接口设置的设备将不会被使用，而是使用环境变量指定的设备。

```
cnrtSetRuntimeContextDeviceId(cnrtRuntimeContext_t pctx, int dev_ordinal)
```

初始化 Context

```
cnrtInitRuntimeContext(cnrtRuntimeContext_t pctx, void *extra)
```

创建 Context 队列

```
cnrtRuntimeContextCreateQueue(cnrtRuntimeContext_t pctx, cnrtQueue_t *queue)
```

调用 Context

该接口会把当前的任务放到传入的队列中，MLU 设备会串行计算队列中的任务。

```
cnrtInvokeRuntimeContext(cnrtRuntimeContext_t pctx, void **params, cnrtQueue_t queue,
↪ void *extra)
```

为 Context 创建 Notifier

```
cnrtRuntimeContextCreateNotifier(cnrtRuntimeContext_t pctx, cnrtNotifier_t␣
↪*pnotifier)
```

Copyright © 2020 Cambricon Corporation.

销毁 Context

```
cnrtDestoryRuntimeContext(cnrtRuntimeContext_t pctx)
```

4.2.2.2 共享 Context 权值和指令内存

在设置神经网络模型运行时，用户可以为每个线程创建一个 Context，使这些 Context 并发运行。这种方法性能最佳，但是由于每个线程都有独立的 Context，会导致占用的内存空间过大。如果想要减小内存开销，用户可以只创建一个 Context。因为每个 Context 调用的权值和指令是相同的，所有线程可以共享一个 Context 权值和指令的内存空间。这种方法能够减少内存占用，并且性能影响较小。

想要使用这种方法，用户可以先创建一个原始 Context，再通过 `cnrtForkRuntimeContext()` 接口复制出一个 Context。复制的 Context 能和原始 Context 共享权值和指令内存空间。另外该接口也会初始化复制的 Context 的中间结果等私有空间，因此用户无需再初始化复制的 Context。用户可以多次调用该接口来复制多个 Context，所有复制的 Context 都会与原始 Context 共享权值和指令内存。并且复制的 Context 和原始 Context 可以分别传给不同的线程做并发运行。

`cnrtForkRuntimeContext()` 接口需在调用 `cnrtInitRuntimeContext()` 接口初始化 Context 后，并且在调用 `cnrtInvokeRuntimeContext()` 接口前调用。需要注意的是，用户需要调用 `cnrtDestroyRuntimeContext()` 接口销毁 Context。

相关示例，请查看共享 Context 权值和指令内存示例。

4.2.2.3 离线模式示例

```
cnrtInit(0);
//load model and extract function
cnrtModel_t model;
cnrtLoadModel(&model,name);
cnrtFunction_t function;
cnrtCreateFunction(&function);
cnrtExtractFunction(&function,model,name);


...
//Get I/O data description from the function
int inputNum, outputNum;
int64_t *inputSizeS, *outputSizeS;
cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);


cnrtRuntimeContext_t ctx;
```

```
cnrtCreateRuntimeContext(&ctx, op_function, nullptr);
cnrtSetRuntimeContextDeviceId(ctx,card_ordinal);

cnrtRuntimeContextCreateQueue(ctx,&queue);
cnrtRuntimeContextCreateNotifier(ctx,&eventBegin);
cnrtRuntimeContextCreateNotifier(ctx,&eventEnd);

//prepare I/O data on MLU
for (int i = 0; i < inputNum; i++) {
        // malloc mlu memory
        cnrtMalloc(&(inputMluPtrS[i]), inputSizeS[i]);
        cnrtMemcpy(inputMluPtrS[i], inputCpuPtrS[i], inputSizeS[i], CNRT_MEM_TRANS_DIR_
↪HOST2DEV);
}
for (int i = 0; i < outputNum; i++) {
        // malloc mlu memory
        cnrtMalloc(&(outputMluPtrS[i]), outputSizeS[i]);
}

cnrtPlaceNotifier(eventBegin, queue);
cnrtInvokeRuntimeContext(ctx, param, queue, NULL);
cnrtPlaceNotifier(eventEnd,queue);

cnrtSyncQueue(queue);

// copy mlu result to cpu
for (int i = 0; i < outputNum; i++) {
        cnrtMemcpy(outputCpuPtrS[i], outputMluPtrS[i], outputSizeS[i], CNRT_MEM_TRANS_DIR_
↪DEV2HOST);
}


//Destroy the resources
cnrtDestoryQueue(queue);
cnrtDestoryRuntimeContext(ctx);
cnrtDestroyFunction(function);
cnrtUnloadModel(model);
cnrtDestroy();
```

Copyright © 2020 Cambricon Corporation.

4.3 功能

4.3.1 功能介绍

CNRT 主要包含以下功能：

- **设备管理**：提供了管理设备接口。如：设备初始化、设备查询、设备指定等。
- **内存管理**：提供了内存管理接口。如：内存分配、内存释放、内存拷贝等。
- **队列**：提供了队列相关接口。如：队列的创建、任务下发、销毁等。
- **Notifier**：提供了统计硬件执行时间相关接口。
- **执行控制**：提供了控制 MLU 计算的接口，支持异步、并发、调度等基本功能。
- **离线模型运行**：脱离 CNML，基于 CNRT 单独运行。

4.3.2 设备管理

CNRT 设备管理模块提供了管理设备的 API，如设备初始化、设备查询、设备指定等。设备管理主要分为三部分：设备初始化、设备操作、设备销毁。

4.3.2.1 设备初始化

在当前系统中初始化 CNRT 运行环境。调用下面 API 初始化设备。当 `flags` 参数设置为 0 时，则初始化真实设备，主要用于在线或者离线运行。同时，初始化该设备的进程默认使用设备 0。当 `flags` 参数设置为 1 时，则初始化假设备，主要用于 CNML 编译。

```
cnrtInit(unsigned int flags);
```

该 API 可在进程开始时调用一次，也可在每个线程开始时调用，但需要在使用设备资源前调用此方法。调用要和 `cnrtDestroy()` API 调用次数相匹配。如整个进程在开始调用一次 `cnrtInit()` API，则在进程结束时调用一次 `cnrtDestroy()` API。如在每个线程开始时调用一次 `cnrtInit()` API，则在每个线程结束时需要调用一次 `cnrtDestroy()` API。

> **注意：**
>
> 在调用任何 CNRT API 之前需要调用此 API 初始化环境，保证线程安全。

4.3.2.2 设备操作

获取系统中的 MLU 设备个数

获取系统中的 MLU 设备个数。API 如下所示，其中 `dev_num` 输出参数的值为设备数目。

Copyright © 2020 Cambricon Corporation.

可以调用此 API，获取设备号。设备号取值范围应在 [0, cnrtGetDeviceCount() – 1]。

```
cnrtGetDeviceCount(unsigned int *dev_num);
```

获取设备句柄

通过给定的设备序号获取设备句柄。

```
cnrtGetDeviceHandle(cnrtDev_t *pdev, int ordinal);
```

> **注意：**
>
> 调用此 API 之前必须进行设备初始化，且设备号范围要在 [0, cnrtGetDeviceCount() – 1]。

设置设备

设置当前进程或线程上下文所使用的设备。设置后，进程或线程后续所有 MLU 的 API 调用，都会在该指定设备上执行。目前支持三种方法来设置使用的设备。

- 初始化设备时，系统自动设置默认设备。当调用 `cnrtInit()` API 初始化设备时，调用该 API 的进程默认使用设备 0 进行操作。该进程的所有线程也使用设备 0 进行操作。用户可以使用以下其他两种方法修改进程或线程使用的设备。
- 调用 `cnrtSetCurrentDevice()` API 设置某一线程使用的设备。用户调用该 API 后，CNRT 内部为此线程维护一个指定设备。

  > **注意：**
  > 用户需要在调用所有设备接口之前调用 `cnrtSetCurrentDevice()` API。

  设置方法：首先调用 `cnrtInit()` API 初始化设备，再调用 `cnrtGetDeviceHandle()` API 获取设备句柄。最后调用 `cnrtSetCurrentDevice()` API 将获得的句柄设为当前线程的使用设备。
- 通过CNRT_DEFAULT_DEVICE 环境变量来指定使用的设备。在同一进程下，设置该进程的所有线程使用同一个设备进行操作。该进程创建的所有子进程、所有线程将只使用该指定设备。同时，通过 `cnrtSetRuntimeContextDeviceId()` 指定的设备不再使用，而是使用该环境变量指定的设备。此方法避免用户对每一个线程重复操作来设置同一个设备。设置 `CNRT_DEFAULT_DEVICE` 后，调用 `cnrtInit()` API 就可以直接操作设备，进行 MLU 上的计算。

系统默认优先使用 `CNRT_DEFAULT_DEVICE` 环境变量指定的设备进行操作。当未找到环境变量指定的设备时，使用 `cnrtSetCurrentDevice()` API 指定的设备。如果没有找到该 API 指定的设备时，则使用 `cnrtInit()` API 初始化设备时设置的默认设备 0。

获取当前线程设备句柄

使用下面接口获取当前线程设备句柄。调用该接口前，需调用 `cnrtSetCurrentDevice()` API 来设置当

前线程使用的设备。

```
cnrtGetCurrentDevice(cnrtDev_t *pdev);
```

4.3.2.3 设备销毁

此方法用于销毁设备资源，如释放设备内存、关闭设备。调用此 API 前，要调用 `cnrtInit()` API 来进行设备初始化。

该 API 可在进程退出时调用一次，也可在每个线程退出时去调用，但调用次数要和 `cnrtInit()` API 调用次数相匹配。在不需要设备资源时，调用此 API。如整个进程在开始调用一次 `cnrtInit()` API，则在进程结束时调用一次 `cnrtDestroy()` API。如在每个线程开始时调用一次 `cnrtInit()` API，则在线程结束时调用一次 `cnrtDestroy()` API。

```
cnrtDestroy(void);
```

警告:

最后调用此 API 将销毁设备上的所有资源包括关闭设备。

4.3.2.4 示例

单线程

```
cnrtDev_t pdev;
cnrtInit(0);
cnrtGetDeviceHandle(&dev, 0);
cnrtSetCurrentDevice(dev);
    .
    .
    .
cnrtDestroy();
```

多线程

```
ing main() {
  cnrtInit(0);
  cnrtGetDeviceHandle(&dev, 0);
  cnrtSetCurrentDevice(dev);
      .
      .
```

Copyright © 2020 Cambricon Corporation.

```
      .
  thread_param[i].ordinal = i;
  for (i = 0; i < n; i++)
    pthread_create(&tid[i], NULL, &thread_func, &thread_param[i]);
      .
      .
      .
  cnrtDestroy();
}
void thread_fun(void *args) {
  cnrtInit(0);
  ordinal = thread_param.ordinal;
  cnrtGetDeviceHandle(&dev, ordinal);
  cnrtSetCurrentDevice(dev);
    .
    .
    .
  cnrtDestroy();
}
```

4.3.3 内存管理

内存管理主要分为主机端内存管理、MLU 端内存管理和主机端与 MLU 端内存相互拷贝。

4.3.3.1 主机端内存管理

CNRT 提供主机端内存分配和释放接口。

内存分配

在主机上分配 bytes 大小的空间，并将 pPtr 参数指向分配的空间。分配成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtMallocHost(void **pPtr, size_t bytes, cnrtMemType_t type);
```

内存释放

释放主机内存。释放成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtRet_t cnrtFreeHost(void *ptr);
```

Copyright © 2020 Cambricon Corporation.

4.3.3.2 MLU 端内存管理

CNRT 提供了 MLU 端内存分配、初始化、释放、获取和设置栈空间的接口。

内存分配

分配内存到指定空间

为 MLU 内存分配指定空间。分配成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtRet_t cnrtMalloc(void **pPtr, size_t bytes);
```

为 P2P 分配内存空间

为 P2P（Peer-to-Peer）分配 MLU 端内存空间。分配成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtRet_t cnrtMallocFrameBuffer(void **pPtr, size_t bytes);
```

申请不同类型内存

CNRT 扩展内存申请接口，用于申请不同类型的内存。分配成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtRet_t cnrtMallocBufferEx(void **pPtr, void *param);
```

内存初始化

将 `ptr` 参数指向的 `bytes` 大小的 MLU 端内存空间，初始化为 `c`。初始化成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtRet_t cnrtMemset(void *ptr, int c, size_t bytes);
```

内存释放

释放指定的内存空间

释放 `ptr` 参数指向的 MLU 端的内存空间。释放内存空间时，用户无需指定设备。释放成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtRet_t cnrtFree(void *ptr);
```

释放指定的内存空间数组

释放 ptr 指向的 MLU 端内存空间数组。释放成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtRet_t cnrtFreeArray(void **ptr, int length);
```

获取和设置栈空间

设置栈空间大小

设置 MLU 设备栈空间大小，单位 MB。获取成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtRet_t cnrtSetLocalMem(unsigned int local_mem_size);
```

获取栈空间大小

获取 MLU 设备栈空间大小，单位 MB。获取成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

```
cnrtRet_t cnrtGetLocalMem(unsigned int *pLocalsize);
```

4.3.3.3 MLU 与主机间的内存拷贝

同步拷贝

同步拷贝主机端与 MLU 端之间的数据。同步拷贝方式适用于数据量较小时。如果数据量比较大，用户可以考虑使用异步拷贝方式。

从地址 src 拷贝 bytes 数据到地址 dst，dir 参数用来确定由主机端拷贝到 MLU 端还是 MLU 端拷贝到主机端。

```
cnrtRet_t cnrtMemcpy(void *dst, void *src, size_t bytes, cnrtMemTransDir_t dir);
```

异步拷贝

异步拷贝 MLU 端与主机端之间的数据。适用于拷贝数据量较大时。

为了提高 MLU core 利用率，用户需要创建至少两个线程，一个线程准备数据，另一个线程执行计算。如果使用异步拷贝，用户只需创建一个线程来完成数据准备和计算。此外，数据拷贝和计算可以并行完成，无需等待前一份数据计算完成后，再进行下一份数据的拷贝。因此用户线程接口不会再阻塞。

```
cnrtRet_t cnrtMemcpyAsync(void *dest_addr, void *src_addr, size_t bytes, cnrtQueue_t queue,␣
↪cnrtMemTransDir_t dir)
```

示例

查看异步拷贝示例程序 。

4.3.4 队列

在 CNRT 中，一个队列（cnrtQueue_t）是由主机端发布的一系列对 MLU 设备的执行操作。所有的设备操作都是通过队列进行的，其中包括计算任务、通讯任务以及事件等。同一个队列可以容纳多个任务。在一个队列中，操作是按顺序串行执行的，不同的队列中的操作可以并发执行。队列内部无论是执行 Notifier 任务还是计算任务，始终遵循 FIFO（First In First Out，先进先出）调度原则。当没有指定队列的时候，CNRT 会使用默认的队列。

MLU270 和 MLU220 单卡队列的数量上限为 4,096。

CNRT 使用同步机制来监控一个队列中所有任务是否执行完毕，详情请查看同步机制。

4.3.4.1 队列的属性

队列具有以下属性：

- 串行性：下发到同一个队列中的任务，按照下发顺序串行执行，使用先进先出原则。
- 异步性：任务下发到队列是一个异步的过程。下发完成后程序的控制队列会回到主机侧，主机程序可以继续往下执行。CNRT 提供队列的同步接口 `cnrtSyncQueue()` ，用于等待整个队列中的所有任务完成。任务的同步需要用户主动调用同步接口发起。

如果想要任务之间并行执行，可以创建多个队列并将任务分配到不同的队列中。

单个队列，先进先出示例：

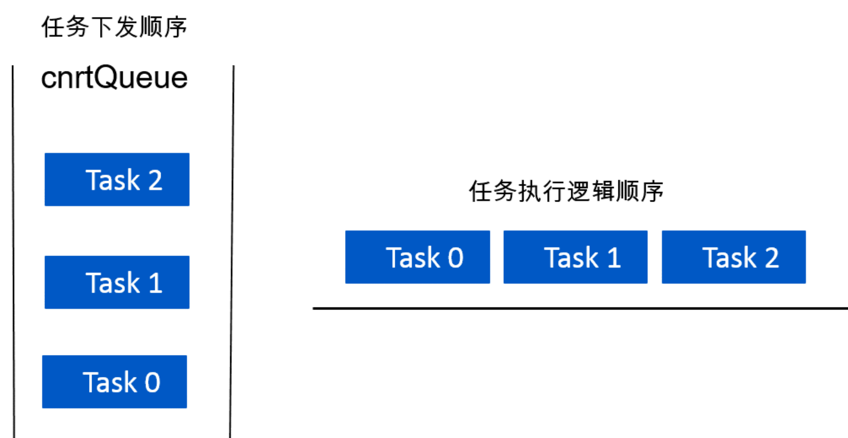

图 4.2: 单个队列任务执行机制

多个队列，并发处理示例：

Copyright © 2020 Cambricon Corporation.

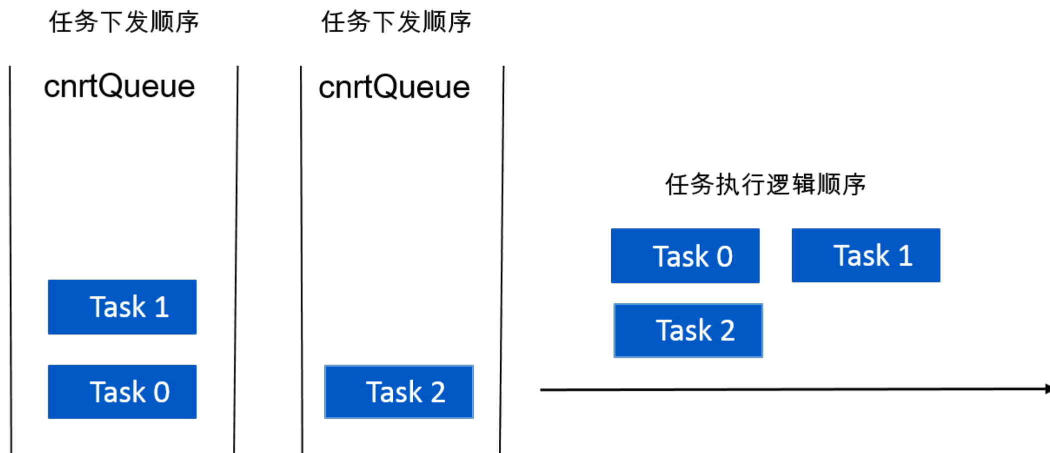

图 4.3: 多个队列任务执行机制

4.3.4.2 接口介绍

队列的使用主要通过以下接口完成：

创建队列

下面 API 用于创建一个队列。该 API 可被多次调用来创建多个队列。

```
cnrtCreateQueue(cnrtQueue_t *queue);
```

创建队列调用 API 的次数和 `cnrtInvokeRuntimeContext_V2()` API 次数要相同。如果创建了多个队列，也需要多次调用 `cnrtInvokeRuntimeContext_V2()`，分别传入不同的队列，才能起到任务并行的效果。如果多次调用 `cnrtInvokeRuntimeContext_V2()`，但传入的都是同一个队列，则不会提高执行效率。

同步队列

下面 API 同步队列中的任务，等待所有的任务执行完成。该 API 要在 `cnrtInvokeRuntimeContext_V2()` 之后调用，因为 `cnrtInvokeRuntimeContext_V2()` 是异步的，所以需要一个同步的机制去等待队列所有任务执行完成后再退出程序。

```
cnrtSyncQueue(cnrtQueue_t queue);
```

销毁队列

下面 API 销毁创建的队列。一般在进程退出或者线程退出的时候执行此 API。调用此方法需要和 `cnrtCreateQueue()` 配对使用。即调用一次 `cnrtCreateQueue()`，在线程退出时调用一次 `cnrtDestroyQueue()`。

Copyright © 2020 Cambricon Corporation.

```
cnrtDestroyQueue(cnrtQueue_t queue);
```

4.3.4.3 示例

队列使用 API：

```
cnrtQueue_t queue;
cnrtCreateQueue(&queue);
cnrtInvokeRuntimeContext(ctx, param, queue, NULL);
cnrtSyncQueue(queue);
cnrtDestroyQueue(queue);
```

4.3.5 同步机制

同步机制用于监控队列中所有任务是否执行完毕，并基于任务队列的核心思想，同步队列中执行的任务。在 MLU270 和 MLU220 中，当 MLU 计算任务下发到队列，用户需要调用 `cnrtInvokeRuntimeContext_V2()` API 来执行队列任务，而这个 API 的执行是异步的。因此，需要一个同步的机制去等待队列中所有任务执行完，再将最终结果拷贝到主机端。

任务队列核心思想是把希望串行执行的任务放到同一个队列内，把希望并行执行的任务放到不同的队列内。详情如下：

- 任务下发到队列后异步执行。
- 同一个队列内的任务按下发的先后顺序串行执行。
- 不同队列之间的任务并行执行。

4.3.5.1 同步机制执行流程

使用同步机制执行任务的主要步骤如下：

1. 调用 `cnrtCreateQueue()`，创建队列。
2. 调用 `cnrtInvokeRuntimeContext_V2()`，将任务下发到队列。
3. 调用 `cnrtSyncQueue()`，同步队列中的任务。
4. 调用 `cnrtDestroyQueue()`，销毁队列。

4.3.6 任务执行

4.3.6.1 任务执行流程

任务的执行可以脱离 CNML，直接在 CNRT 上运行。在准备好输入数据并加载离线模型后，通过设置函数 API 启动参数，将任务下发到 MLU core 上进行计算。待 MLU 计算完成后，将数据拷贝到主机端。

具体流程如下：

1. 通过 `cnrtGetInputDataSize()` 和 `cnrtGetOutputDataSize()` API 获得输入、输出数据内存的大小。用户还可以调用 `cnrtGetInputDataType()` 和 `cnrtGetOutputDataType()` API 获得输入输出数据的类型。调用 `cnrtGetInputDataShape()` 和 `cnrtGetOutputDataShape()` API 获得输入输出数据的维度。用户还可以调用 API，在预处理时，设置数据量化和数据转置，例如 float32 转为 float16，张量维度顺序从 NCHW 转为 NHWC。
2. 调用 `cnrtMalloc()` API，分配 MLU 输入和输出数据的内存空间。
3. 调用 `cnrtMemcpy()` API，将输入数据从主机端上拷贝到 MLU。
4. 声明 `cnrtRuntimeContext_t` 参数。
5. 调用 `cnrtCreateRuntimeContext()` API，创建 Context。
6. 调用 `cnrtSetRuntimeContextDeviceId()` API，设置 Context 设备 ID。
7. 调用 `cnrtInitRuntimeContext()` API，初始化 Context。
8. 调用 `cnrtRuntimeContextCreateQueue()` API，创建队列。
9. 调用 `cnrtInvokeRuntimeContext_V2()` API，调用队列。
10. 调用 `cnrtSyncQueue()` API，同步队列。
11. 调用 `cnrtMemcpy()` API，把 MLU 计算结果拷贝回主机端。
12. 释放相关内存空间。

4.3.6.2 示例

查看任务执行示例程序。

4.3.7 Notifier

CNRT 使用 Notifier 机制统计硬件执行时间。

4.3.7.1 Notifier 机制

Notifier 是一种特殊的任务，它和计算任务一样可以放置到队列中执行。相比计算任务，Notifier 不执行实际的硬件操作。如果相邻的两个或者多个 Notifier 之间包含计算任务，可以通过 Notifier 来实现对计算任务耗时的精确统计。

如下图所示，当驱动调度到 `notifier_begin` 任务时，`notifier_begin` 会被记录一个时间戳，这个时间戳是当前队列中执行过的所有计算任务的累计时间。驱动调度完 `notifier_begin` 之后会立即调度计算任务。在计算任务执行完成后，驱动会调度 `notifier_end` ，记录当前队列中执行过的所有计算任务的累计时间。通过对两个 Notifier 之间的时间戳做差来获得计算任务的执行时间。

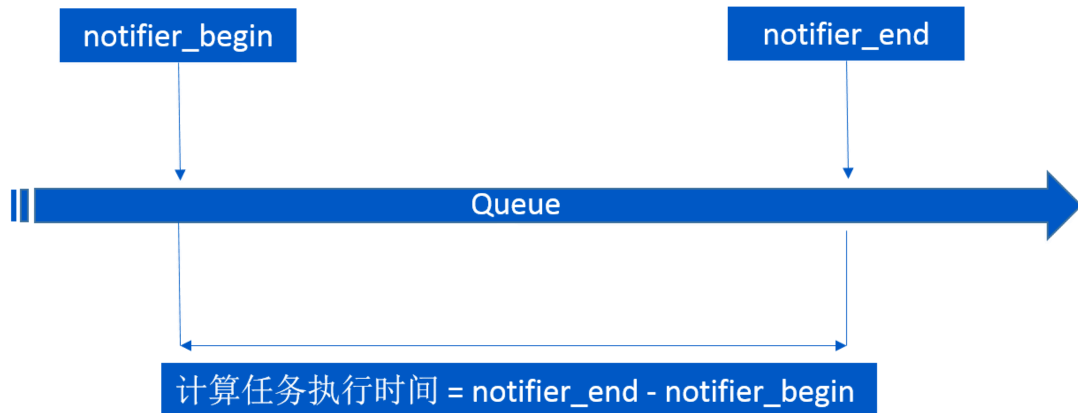

图 4.4: Notifier 执行方法

4.3.7.2 Notifier 计时步骤

Notifier 计时步骤如下：

1. 调用 cnrtRuntimeContextCreateNotifier() 接口，创建 notifier_begin 和 notifier_end 两个 Notifier，用于统计硬件执行时间。
2. 在队列中加入计算任务之前，调用 cnrtPlaceNotifier() API 设置 notifier_begin。
3. 调用 cnrtInvokeRuntimeContext() API 将任务下发到队列。
4. 在队列中加入任务之后，调用 cnrtPlaceNotifier() API 设置 notifier_end。
5. 确保队列中的计算任务执行完成后，调用 cnrtNotifierDuration() API 获取硬件执行时间。用户可以调用 cnrtSyncQueue() API 来监控队列，确保任务完成。

4.3.7.3 示例

Notifier 计时操作所调用的 API 示例如下：

```
float time_elapsed;
cnrtNotifier_t notifier_begin;
cnrtRuntimeContextCreateNotifier(...,&notifier_begin);
cnrtNotifier_t notifier_end;
cnrtRuntimeContextCreateNotifier(...,&notifier_end);
cnrtPlaceNotifier(notifier_begin, queue);
cnrtInvokeRuntimeContext(..., ..., queue, ...);
cnrtPlaceNotifier(notifier_end, queue);
cnrtSyncQueue(queue);
cnrtNotifierDuration(notifier_begin, notifier_end, &time_elapsed);
```

4.3.8 离线模型管理

离线模型是一个序列化的、经过编译的网络或算子（基本算子或融合算子）的文件。CNRT 通过加载离线模型文件来驱动 MLU core 完成计算。寒武纪离线模型包含模型版本、Core Version（MLU Core 版本）信息、MLU 指令、权值数据、输入输出数据规模、参数信息等。离线模型文件通过 CNML 生成，而离线模型的运行可以脱离 CNML，基于 CNRT 单独运行。由于脱离了上层软件栈，离线模型的执行具有更好的性能和通用性。

4.3.8.1 功能介绍

加载模型到内存

下面 API 用于将离线模型文件加载到内存。

```
cnrtLoadModel(cnrtModel_t *pmodel, const char *fname);
```

> **注意:**
>
> 调用此 API 之前，需要有已经生成好的离线模型。

生成 Function

下面 API 从加载的模型中生成 Function，用于 MLU 的计算。可以根据需要生成一份或多份 Function。

```
cnrtExtractFunction(cnrtFunction_t *pfunction, cnrtModel_t model,
                    const char *symbol);
```

每个线程需要调用这个 API 生成一份 Function。此 Function 线程是独有的，因为每个线程的操作不一样，需要的信息也不一样。`symbol` 符号的值要和生成的离线模型的标志一样，如生成的模型类型是“mlp”，“fusion”等。

获取 Function 个数

下面 API 用于获取模型中 Function 的个数。

```
cnrtGetFunctionNumber(cnrtModel_t model, int *func_num);
```

获取模型并行度

下面 API 用于获取模型中的模型并行度，即 mp（model parallelism）。mp 用于处理单模型以及单输入的 MLU 核心数运算。

Copyright © 2020 Cambricon Corporation.

```
cnrtQueryModelParallelism(cnrtModel_t model, int *modelParallelism);
```

获取本地存储器最大容量值

获取模型中容量最大的本地存储器（localMem）的容量值。localMem 是每个核独有的，用于存放临时中间结果。

```
cnrtRet_t cnrtQueryModelLocalMemSize(cnrtModel_t model, uint64_t *local_mem_size);
```

用户可根据需要调用 `cnrtSetLocalMem()` API 来设置每个核的 localMem 大小。

释放资源

释放模型占用的 CPU 资源。

```
cnrtRet_t cnrtUnloadModel(cnrtModel_t model);
```

4.3.8.2 模型加载流程

模型版本采用 Model Level 概念描述，CNML 和 CNRT 库都有支持的最高 Model Level 等级，CNML 和 CNRT 目前默认支持小于等于最高等级 Model Level 的模型生成和加载。

离线模型加载流程如下：

1. 创建 `cnrtModel_t` 结构体。
2. 调用cnrtLoadModel() API 将离线模型文件中的数据写入到创建的 `cnrtModel_t` 结构体变量中。
3. 调用cnrtGetFunctionNumber() API 获取模型中包含的 Function 数量。
4. 调用 `cnrtCreateFunction()` API 初始化 `cnrtFunction_t` 结构体变量。
5. 调用cnrtExtractFunction() API 提取模型中的模型信息，并将其写入上一步创建的 `cnrtFunction_t` 结构体变量中。

4.3.8.3 模型版本兼容性

模型版本

模型版本，即 Model Level，用来表示模型版本信息，区分 CNML 和 CNRT 版本号。因为 CNML 和 CNRT 版本变更频繁，模型 Model Level 变更相对低频且趋于稳定。

CNML 和 CNRT 提供接口反馈支持的最低和最高 Model Level。

版本演进，兼容性原则

随着 CNML 和 CNRT 库不断演进，离线模型也会持续升级，兼容性采用 CNRT 库向前兼容旧版本离线模型，即高版本 CNRT 库仍然可以支持旧版本的离线模型。但新版本的离线模型不保证可以被所有旧版本的 CNRT 库解析。

Copyright © 2020 Cambricon Corporation.

4.3.8.4 示例

离线模型管理示例如下：

```
cnrtInit(0);
unsigned dev_num;
cnrtGetDeviceCount(&dev_num);
if (dev_num == 0)
return NULL;
cnrtDev_t dev;
cnrtGetDeviceHandle(&dev, 0);
cnrtSetCurrentDevice(dev);

// load model and get function
cnrtModel_t model;
char fname[100] = "";
// The name parameter represents the name of the offline model file. It is also the name of a␣
↪function in the offline model file.
strcat(fname, (char *)name);
strcat(fname, ".mef");
printf("load file: %s\n", fname);
cnrtLoadModel(&model, fname);

int64_t totalMem;
cnrtGetModelMemUsed(model, &totalMem);
printf("total memory used: %" PRId64 " Bytes\n", totalMem);
int func_num = 0;
cnrtGetFunctionNumber(model, &func_num);
printf("model function number is %d\n", func_num);

// setup function
cnrtFunction_t function;
cnrtCreateFunction(&function);
cnrtExtractFunction(&function, model, (char *)name);

uint64_t local_mem_size;
cnrtQueryModelLocalMemSize(model, &local_mem_size);
printf("model kernels' biggest localMem size is: %" PRId64 " MB\n", local_mem_size);
```

Copyright © 2020 Cambricon Corporation.

Cambricon
寒 武 纪

5 编程模型

5.1 MLU220 编程模型

与 MLU270 相比，MLU220 共 4 个 core，1 个 cluster，所以编程模型与 MLU270 略有不同。由于 MLU220 EDGE 的边缘侧设备 ARM CPU 与 MLU 共享 DDR，内存数据读写方式也会与 MLU270 有所不同。

5.1.1 MLU220 数据读写优化

由于 MLU220 EDGE 的边缘侧设备 ARM CPU 与 MLU 共享 DDR，为了减少 CPU 内存从 MLU 内存拷贝数据的次数，提高性能，用户可以调用 `cnrtMap()` API 将设备内存地址映射到 CPU。此外，如果只想对内存中的某一块区域做内存地址映射，用户可以调用 `cnrtMapRange()` API 来完成。但是 `cnrtMapRange()` API 只用于通过 `cnrtMalloc()` API 分配的内存做映射。

将 MLU 内存映射到 CPU 后，用户可以调用 `cnrtCacheOperation()` API 完成数据同步，保证数据读写的一致性。此外，如果只想对映射的地址中的某一段数据做数据同步，可以通过调用 `cnrtCacheOperationRange()` API 来完成。相关 API 详情，请参考《寒武纪 CNRT 开发者手册》。

注意:

如果调用 `cnrtMap()` API，用户后续必须调用 `cnrtUnmap()` API 取消映射。

5.1.1.1 CPU 写数据到内存

当 CPU 写数据到内存后，MLU 读取数据，用户执行流程如下：

1. CPU 将数据写入内存。
2. 调用 `cnrtCacheOperation()` 或者 `cnrtCacheOperationRange()` API，来同步 CPU 和 MLU 数据，保证 CPU 与 MLU 读写数据一致性。API 中 `opr` 变量设置值为 **FLUSH**。
3. MLU 从内存读取数据。

5.1.1.2 MLU 写入数据到内存

当 MLU 写入数据到内存后，CPU 读取数据，用户执行流程如下：

1. 调用 `cnrtCacheOperation()` 或者 `cnrtCacheOperationRange()` API，其中 `opr` 变量设置值为 **FLUSH**，保证数据的准确性。
2. MLU 将数据写入内存，并完成相关任务，如数据计算。这步完成前，CPU 不可以读写 cache 操作的那块地址。
3. CPU 从内存读取数据。

5.2 指定 DDR 通道和 Cluster 来执行 Cluster 任务

当多个 cluster 任务并行执行时，内存默认将 cluster 任务全部分配到一个 DDR 通道上，而不是分配到 MLU270 的 4 个 DDR 通道上。从而导致多个任务同时访问一个 DDR 通道，造成访存冲突，板卡利用率降低，带来访问延迟。为了提高性能，用户可以通过指定使用的 DDR 通道和访存的 DDR cluster，来提高访存效率。

在 MLU270 上，一个线程运行一个 cluster 任务。对于每个 cluster 任务，用户需要在申请输入输出内存以及初始化 runtime context 之前，指定 DDR 通道，并在运行时设置亲和性对应的 cluster。支持的 DDR 通道可以通过 `cnrtChannelType_t` 来设置，并通过调用 `cnrtSetCurrentChannel()` API 来完成。每个 cluster 使用一个比特位表示，即 `0x01` 表示 cluster0，`0x02` 表示 cluster1，`0x04` 表示 cluster2，`0x08` 表示 cluster3。用户可以在 `cnrtInvokeParam_t` 中设置亲和性和对应的 cluster，并通过调用 `cnrtInvokeRuntimeContext_V2()` API 来完成。

5.2.1 示例代码

编程示例请参考指定 DDR 通道和 Cluster 来执行 Cluster 任务 。

5.3 BANG C 混合编程

CNRT 将 BANG C 语言生成的指令和 CNML 的执行逻辑统一起来，从而支持 CNML 的特性及多种运行模式，如在线、离线，逐层、融合等。

本节主要介绍了相关接口和接口使用示例。关于如何使用 BANG C 编程，请查看《寒武纪 BANG C 开发者手册》中“Programming Model”章节。

5.3.1 功能接口介绍

CNRT 提供了参数操作、数据占位和调用 kernel 的接口。接口运行成功返回 CNRT_RET_SUCCESS，否则返回相应的错误码。

5.3.1.1 参数操作

获取 kernel 中的 params 参数

```
cnrtRet_t cnrtGetKernelParamsBuffer(cnrtKernelParamsBuffer_t *params);
```

拷贝 src_params_buf 到 dst_params_buf

```
cnrtCopyKernelParamsBuffer(cnrtKernelParamsBuffer_t dst_params_buf,
                           cnrtKernelParamsBuffer_t src_params_buf);
```

向 cnrtKernelParamsBuffer_t 中增加一个常量参数

```
cnrtKernelParamsBufferAddParam(cnrtKernelParamsBuffer_t params, void *data, size_t bytes);
```

销毁 cnrtKernelParamsBuffer_t 变量

```
cnrtRet_t cnrtDestroyKernelParamsBuffer(cnrtKernelParamsBuffer_t params);
```

5.3.1.2 数据占位

为输入数据占位

为输入数据占位，以便运行时填入输入数据的指针地址。

```
cnrtRet_t cnrtKernelParamsBufferMarkInput(cnrtKernelParamsBuffer_t params);
```

为输出数据占位

为输出数据占位，以便运行时填入输出数据的指针地址。

```
cnrtRet_t cnrtKernelParamsBufferMarkOutput(cnrtKernelParamsBuffer_t params);
```

为常量数据占位

为常量数据占位，以便运行时填入常量数据的指针地址。

```
cnrtRet_t cnrtKernelParamsBufferMarkStatic(cnrtKernelParamsBuffer_t params);
```

为输入输出数据占位

为输入输出数据占位，以便运行时回填拆分后的值。

```
cnrtRet_t cnrtKernelParamsBufferMarkPluginOpDimension(cnrtKernelParamsBuffer_t params, int␣
↪tensor_id, int dim, int increase);
```

5.3.1.3 内核调用

内核调用接口 V2 版

内核调用接口的 v2 版本。通过在 MLU 上给定的参数块，调用接口。

```
cnrtRet_t cnrtInvokeKernel_V2(const void *function, cnrtDim3_t dim,
                              cnrtKernelParamsBuffer_t params,
                              cnrtFunctionType_t func_type,
                              cnrtQueue_t queue);
```

内核调用接口 V3 版

内核调用接口的 v3 版本。相对其他版本，该版本增加 cluster 亲和性的特性，用户可以指定运行该接口的线程，从而提升板卡资源利用率。另外，API 内部操作得到优化，从而性能得到了相应的提升。设置中要注意 DDR 通道要与 cluster 对应。关于 cluster 亲和性详细内容，请查看指定 DDR 通道和 Cluster 来执行 Cluster 任务 。

```
cnrtRet_t cnrtInvokeKernel_V3(const void *function,
                             cnrtKernelInitParam_t init_param,
                             cnrtDim3_t dim,
                             cnrtKernelParamsBuffer_t params,
                             cnrtFunctionType_t func_type,
                             cnrtQueue_t queue,
                             void *extra_param);
```

执行下面操作来调用内核：

1. 调用 `cnrtSetCurrentChannel()` API 指定使用的 DDR 通道。
2. 调用 `cnrtKernelInitParam_t()` API 初始化参数的数据结构。
3. 调用 `cnrtCreateKernelInitParam()` API 创建初始化参数。
4. 调用 `cnrtInitKernelMemory()` API 初始化内核内存空间。
5. 在 `cnrtInvokeParam_t` 中设置亲和性和对应的 cluster，并通过调用 `cnrtInvokeRuntimeContext_V2()` API 来完成。
6. 调用 `cnrtInvokeKernel_V3()` API 调用内核。
7. 调用 `cnrtDestroyKernelInitParamAndMemory()` API 释放内存。

假设有两个 function 分别是 `function1` 和 `function2` 。将这两个 function 放到不同的两个线程中来调用，并放到不同的 cluster 上去运行。示例如下。

```
void thread_func1() {
      ...
      cnrtSetCurrentChannel(CNRT_CHANNEL_TYPE_0);
      cnrtKernelInitParam_t init_param1;
      cnrtCreateKernelInitParam(&init_param1);
      cnrtInitKernelMemory((void *)&function1, init_param1);

      unsigned int affinity1 = 0x01;
```

```
    cnrtInvokeParam_t invoke_param1;
    invoke_param1.invoke_param_type = CNRT_INVOKE_PARAM_TYPE_0;
    invoke_param1.cluster_affinity.affinity = &affinity1;
    ...
    cnrtInvokeKernel_V3((void *)&function1, init_param1, dim, params, func_type, queue,␣
↪(void *)&invoke_param1);
    cnrtInvokeKernel_V3((void *)&function1, init_param1, dim, params, func_type, queue,␣
↪(void *)&invoke_param1);
    cnrtInvokeKernel_V3((void *)&function1, init_param1, dim, params, func_type, queue,␣
↪(void *)&invoke_param1);
    ...
    cnrtDestroyKernelInitParamAndMemory(init_param1);
    ...
}

void thread_func2() {
    ...
    cnrtSetCurrentChannel(CNRT_CHANNEL_TYPE_1);
    cnrtKernelInitParam_t init_param2;
    cnrtCreateKernelInitParam(&init_param2);
    cnrtInitKernelMemory((void *)&function2, init_param2);

    unsigned int affinity2 = 0x02;
    cnrtInvokeParam_t invoke_param2;
    invoke_param2.invoke_param_type = CNRT_INVOKE_PARAM_TYPE_0;
    invoke_param2.cluster_affinity.affinity = &affinity2;
    ...
    cnrtInvokeKernel_V3((void *)&function2, init_param2, dim, params, func_type, queue,␣
↪(void *)&invoke_param2);
    cnrtInvokeKernel_V3((void *)&function2, init_param2, dim, params, func_type, queue,␣
↪(void *)&invoke_param2);
    cnrtInvokeKernel_V3((void *)&function2, init_param2, dim, params, func_type, queue,␣
↪(void *)&invoke_param2);
    ...
    cnrtDestroyKernelInitParamAndMemory(init_param2);
    ...
}
```

Copyright © 2020 Cambricon Corporation.

5.3.2 BANG C 混合编程示例

以下示例介绍了如果通过 BANG C 编程实现 a=a+b 计算。其中 kernel.h 文件为 BANG C 的头文件。kernel.mlu 文件是 BANG C 代码的实现。main.cpp 为 host 端代码，用于调用 BANG C 代码完成计算。

kernel.h 文件代码示例如下：

```
//! \file kernel.h
//! \brief Kernel function defination.

typedef unsigned short half;
#ifdef __cplusplus
extern "C" {
#endif
//! \brief Kernel function: Add
//!         ac[data[index]]++
//! \param ac : atomic_counter
//! \param data : "0,1" data array
//! \param tid : thread index
//! \return : void
void kernel(int *a, int *b);
#ifdef __cplusplus
}
#endif
```

kernel.mlu 文件代码示例如下：

```
#include "mlu.h"

__mlu_entry__ void kernel(int *a, int *b) {
  *a = *a + *b;
}
```

main.cpp 文件代码示例如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "cnrt.h"
#include "kernel.h"

int main(void) {
```

```
  cnrtInit(0);
  cnrtDev_t dev;
  cnrtGetDeviceHandle(&dev, 0);
  cnrtSetCurrentDevice(dev);

  int a, b, out;
  srand((unsigned)time(NULL));
  a = (int)rand();
  b = (int)rand();

  cnrtQueue_t queue;
  cnrtCreateQueue(&queue);

  int *k_a, *k_b;
  cnrtMalloc((void **)&k_a, sizeof(int));
  cnrtMalloc((void **)&k_b, sizeof(int));
  cnrtMemcpy(k_a, &a, sizeof(int), CNRT_MEM_TRANS_DIR_HOST2DEV);
  cnrtMemcpy(k_b, &b, sizeof(int), CNRT_MEM_TRANS_DIR_HOST2DEV);

  cnrtDim3_t dim;
  dim.x = 1;
  dim.y = 1;
  dim.z = 1;
  cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_BLOCK;
  cnrtKernelParamsBuffer_t params;
  cnrtGetKernelParamsBuffer(&params);
  cnrtKernelParamsBufferAddParam(params, &k_a, sizeof(int *));
  cnrtKernelParamsBufferAddParam(params, &k_b, sizeof(int *));
  cnrtInvokeKernel_V2((void *)&kernel, dim, params, func_type, queue);
  cnrtSyncQueue(queue);

  cnrtMemcpy(&out, k_a, sizeof(int), CNRT_MEM_TRANS_DIR_DEV2HOST);
  cnrtFree(k_a);
  cnrtFree(k_b);
  cnrtDestroyKernelParamsBuffer(params);
  cnrtDestroyQueue(queue);
  cnrtDestroy();

  printf("out:%d, a=%d, b=%d, a+b=%d\n", out, a, b, (a+b));
}
```

Cambricon 寒 武 纪

6 调试方法

本章描述 CNRT 的调试方法和工具。

CNRT 提供错误查找接口，帮助用户调试程序。详情请查看《寒武纪 CNRT 开发者手册》中“Error Handling Management”章节。

6.1 使用 loginfo 记录 API 日志

寒武纪提供 logoinfo 功能来记录 CNML API 和 CNRT API 的输入参数，帮助用户在运行出现问题时，能够快速除错，找到问题所在。该功能默认为关闭状态。用户可以通过设置 `NEUWARE_LOGINFO` 全局变量开启或者关闭 loginfo 功能。

输出的日志信息包括 API 函数名、参数信息、参数类型、参数值、函数调用时间、进程 ID、线程 ID、队列、设备地址以及返回值信息。下面为一个日志示例：

```
//一个函数显示多行信息，行前缀为函数运行的进程 ID (tid)。
[28930] Function cnrtMemcpy called;   //显示函数名。
//下面显示该函数的参数信息，包括参数类型以及参数值。每个参数分行显示。
[28930] dest input param: type = void *, value = 0x200000fffff98000;
[28930] src input param: type = void *, value = 0x200000fffff9c000;
[28930] bytes input param: type = size_t, value = 100;
[28930] dir input param: type = cnrtMemTransDir_t, value = 1;
//基础类型的指针（如 int *, float *）会打印其指向的值。
[28930] dev_num input param: type = unsigned int *, value = 4291496;
//枚举类型会直接打印值对应的标识符名。
[28930] dir input param: type = cnrtMemTransDir_t, value = CNRT_MEM_TRANS_DIR_HOST2DEV;
//打印 cnmlTensor_t, cnmlBaseOp_t, cnrtRuntimeContext_t, cnrtDev_t 时，会直接打印相关结构体的具体信息。
[28930] tensor input param: type = cnmlTensor_t, value = { type = CNML_TENSOR  key = <0, ^@> ␣
↪tensor = 0x1719228  newest = (nil) };
[28930] op input param: type = cnmlBaseOp_t, value =
   { core_num = -1  version = CNML_C10
     inputs = (nil)
     outputs = (nil)
```

```
  };
[28930] pctx input param: type = cnrtRuntimeContext_t, value =
     { ref_cnt = 0 dev_ordinal = 0 model_parallelism = 1 channel = CNRT_CHANNEL_TYPE_NONE␣
↪params = (nil) };
[28930] dev input param: type = cnrtDev_t, value = { device_name = MLU270 core_version = CNRT_
↪MLU270 core_num = 16 };

//下面显示函数调用的时间。
[28930] Time: Fri Oct 25 22:09:41 2019
//下面显示进程 ID (process)、线程 ID (thread)、队列 (queue)、设备地址 (device)。
[28930] Process: 28930, Thread: 28930, cnrt_queue: 0x1276d60, cnrt_device: 0x1276c10;
//函数返回时显示下面返回信息，包括函数名、返回值类型和返回值。返回类型为 void 时，不显示返回值。
[28930] Function cnrtLoadModel Return: type = cnrtRet_t, value = CNRT_RET_SUCCESS;
```

如果启用 loginfo 功能，在执行命令前设置全局变量 NEUWARE_LOGINFO 的值为 1。即在命令行输入下面指令。用户需将 command 变量替换为要执行的命令。

```
NEUWARE_LOGINFO=1 ./command
```

例如：~/runtime_api/build/$ NEUWARE_LOGINFO=1 ./runtime_test

运行后，输出日志信息会显示在屏幕上。

Cambricon
寒 武 纪

7 示例程序

7.1 离线模型示例程序

下面示例程序是全连接（mlp）算子的离线模型加载及计算过程。

```
/* Copyright (C) [2019] by Cambricon, Inc. */
/* offline_test */
/*
 * A test which shows how to load and run an offline model.
 * This test consists of one operation --mlp.
 *
 * This example is used for MLU270 and MLU220.
 *
 */

#include "cnrt.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int offline_test(const char *name) {

// This example is used for MLU270 and MLU220. You need to
      choose the corresponding offline model.
  // when generating an offline model, u need cnml and cnrt both
  // when running an offline model, u need cnrt only
  cnrtInit(0);

  // prepare model name
  char fname[100] = "../";
  // The name parameter represents the name of the offline model file.
  // It is also the name of a function in the offline model file.
  strcat(fname, name);
  strcat(fname, ".mef");
```

Copyright © 2020 Cambricon Corporation.

```
    // load model
    cnrtModel_t model;
    cnrtLoadModel(&model, fname);

    cnrtDev_t dev;
    cnrtGetDeviceHandle(&dev, 0);
    cnrtSetCurrentDevice(dev);

    // get model total memory
    int64_t totalMem;
    cnrtGetModelMemUsed(model, &totalMem);
    printf("total memory used: %ld Bytes\n", totalMem);
    // get model parallelism
    int model_parallelism;
    cnrtQueryModelParallelism(model, &model_parallelism);
    printf("model parallelism: %d.\n", model_parallelism);

    // load extract function
    cnrtFunction_t function;
    cnrtCreateFunction(&function);
    cnrtExtractFunction(&function, model, name);

    int inputNum, outputNum;
    int64_t *inputSizeS, *outputSizeS;
    cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
    cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);

    // prepare data on cpu
    void **inputCpuPtrS = (void **)malloc(inputNum * sizeof(void *));
    void **outputCpuPtrS = (void **)malloc(outputNum * sizeof(void *));

    // allocate I/O data memory on MLU
    void **inputMluPtrS = (void **)malloc(inputNum * sizeof(void *));
    void **outputMluPtrS = (void **)malloc(outputNum * sizeof(void *));

    // prepare input buffer
    for (int i = 0; i < inputNum; i++) {
          // converts data format when using new interface model
          inputCpuPtrS[i] = malloc(inputSizeS[i]);
          // malloc mlu memory
          cnrtMalloc(&(inputMluPtrS[i]), inputSizeS[i]);
          cnrtMemcpy(inputMluPtrS[i], inputCpuPtrS[i], inputSizeS[i], CNRT_MEM_TRANS_DIR_
↪HOST2DEV);
```

```
    }
    // prepare output buffer
    for (int i = 0; i < outputNum; i++) {
          outputCpuPtrS[i] = malloc(outputSizeS[i]);
          // malloc mlu memory
          cnrtMalloc(&(outputMluPtrS[i]), outputSizeS[i]);
    }

    // prepare parameters for cnrtInvokeRuntimeContext_V2
    void **param = (void **)malloc(sizeof(void *) * (inputNum + outputNum));
    for (int i = 0; i < inputNum; ++i) {
          param[i] = inputMluPtrS[i];
    }
    for (int i = 0; i < outputNum; ++i) {
          param[inputNum + i] = outputMluPtrS[i];
    }

    // setup runtime ctx
    cnrtRuntimeContext_t ctx;
    cnrtCreateRuntimeContext(&ctx, function, NULL);

    // bind device
    cnrtSetRuntimeContextDeviceId(ctx, 0);
    cnrtInitRuntimeContext(ctx, NULL);

    // compute offline
    cnrtQueue_t queue;
    cnrtRuntimeContextCreateQueue(ctx, &queue);

    // invoke
    cnrtInvokeRuntimeContext_V2(ctx, NULL, param, queue, (void *)&invoke_param);

    // sync
    cnrtSyncQueue(queue);

    // copy mlu result to cpu
    for (int i = 0; i < outputNum; i++) {
          cnrtMemcpy(outputCpuPtrS[i], outputMluPtrS[i], outputSizeS[i], CNRT_MEM_TRANS_DIR_
↪DEV2HOST);
    }

    // free memory space
```

```
    for (int i = 0; i < inputNum; i++) {
          free(inputCpuPtrS[i]);
          cnrtFree(inputMluPtrS[i]);
    }
    for (int i = 0; i < outputNum; i++) {
          free(outputCpuPtrS[i]);
          cnrtFree(outputMluPtrS[i]);
    }
    free(inputCpuPtrS);
    free(outputCpuPtrS);
    free(param);

    cnrtDestroyQueue(queue);
    cnrtDestroyRuntimeContext(ctx);
    cnrtDestroyFunction(function);
    cnrtUnloadModel(model);
    cnrtDestroy();

    return 0;
  }

  int main() {
    printf("mlp offline test\n");
    offline_test("mlp");
    return 0;
  }
```

7.2 异步拷贝示例程序

下面示例程序展示了如何在 MLU 端与主机端异步拷贝数据。

```
/* Copyright (C) [2019] by Cambricon, Inc. */
    /* memcpy_async_test */
    /*
     * A test which shows how to load and run an offline model.
     * This test consists of one operation --mlp.
     *
     */

    #include "tests.h"
```

```
int memcpy_async_test(cnrtCoreVersion_t coreVersion) {
  // when generating an offline model, u need cnml and cnrt both
  // when running an offline model, u need cnrt only
  cnrtInit(0);

  printf("mlp offline test\n");

  // prepare model name
  char name[100];
  if (coreVersion == CNRT_MLU270) {
        strcpy(name, "mlp_mlu270.mef");
  } else if (coreVersion == CNRT_MLU220) {
        strcpy(name, "mlp_mlu220.mef");
  } else {
        printf("Wrong coreVersion !\n");
        return 0;
  }

  char *fname = getDemoCacheModelPath();
  // The name parameter represents the name of the offline model file.
  // It is also the name of a function in the offline model file.
  strcat(fname, name);

  // flag which shows if cnrt function runs normaly
  cnrtRet_t ret = CNRT_RET_SUCCESS;

  // load model
  cnrtModel_t model;
  cnrtLoadModel(&model, fname);

  cnrtDev_t dev;
  cnrtGetDeviceHandle(&dev, 0);
  cnrtSetCurrentDevice(dev);

  // get model total memory
  int64_t totalMem;
  cnrtGetModelMemUsed(model, &totalMem);
  printf("total memory used: %ld Bytes\n", totalMem);
  // get model parallelism
  int model_parallelism;
  cnrtQueryModelParallelism(model, &model_parallelism);
```

```
printf("model parallelism: %d.\n", model_parallelism);

// load extract function
cnrtFunction_t function;
cnrtCreateFunction(&function);
cnrtExtractFunction(&function, model, "mlp");

int inputNum, outputNum;
int64_t *inputSizeS, *outputSizeS;
cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);

// prepare data on cpu
void **inputCpuPtrS = (void **)malloc(inputNum * sizeof(void *));
void **outputCpuPtrS = (void **)malloc(outputNum * sizeof(void *));

// allocate I/O data memory on MLU
void **inputMluPtrS = (void **)malloc(inputNum * sizeof(void *));
void **outputMluPtrS = (void **)malloc(outputNum * sizeof(void *));

// setup runtime ctx
cnrtRuntimeContext_t ctx;
cnrtCreateRuntimeContext(&ctx, function, NULL);

// bind device
cnrtSetRuntimeContextDeviceId(ctx, 0);
cnrtInitRuntimeContext(ctx, NULL);

// compute offline
cnrtQueue_t queue;
cnrtRuntimeContextCreateQueue(ctx, &queue);

// prepare input buffer
for (int i = 0; i < inputNum; i++) {
      // converts data format when using new interface model
      inputCpuPtrS[i] = malloc(inputSizeS[i]);
      // malloc mlu memory
      cnrtMalloc(&(inputMluPtrS[i]), inputSizeS[i]);

      // before using async memcpy, u need to set queue ready
      // Use this interface to keep the copy task from blocking,
      // when the size of copy data too large.
```

```
        ret = cnrtMemcpyAsync(inputMluPtrS[i], inputCpuPtrS[i], inputSizeS[i], queue,
                                            CNRT_MEM_TRANS_DIR_HOST2DEV);
        if (ret == CNRT_RET_SUCCESS) {
          printf("async memcpy host to device success\n");
        } else {
          printf("async memcpy host to device failed!!\n");
          return 0;
        }
    }

    // prepare output buffer
    for (int i = 0; i < outputNum; i++) {
        outputCpuPtrS[i] = malloc(outputSizeS[i]);
        // malloc mlu memory
        cnrtMalloc(&(outputMluPtrS[i]), outputSizeS[i]);
    }

    // prepare parameters for input/output buffers
    void **param = (void **)malloc(sizeof(void *) * (inputNum + outputNum));
    for (int i = 0; i < inputNum; ++i) {
        param[i] = inputMluPtrS[i];
    }
    for (int i = 0; i < outputNum; ++i) {
        param[inputNum + i] = outputMluPtrS[i];
    }

    // invoke
    cnrtInvokeRuntimeContext_V2(ctx, NULL, param, queue, (void *)&invoke_param);

    // copy mlu result to cpu
    for (int i = 0; i < outputNum; i++) {
        // before using async memcpy, u need to set queue ready
        // Use this interface to keep the copy task from blocking,
        // when the size of copy data too large
        ret = cnrtMemcpyAsync(outputCpuPtrS[i], outputMluPtrS[i], outputSizeS[i], queue,
                                            CNRT_MEM_TRANS_DIR_DEV2HOST);
        if (ret == CNRT_RET_SUCCESS) {
          printf("async memcpy dvice to host success\n");
        } else {
          printf("async memcpy device to host failed!!\n");
          return 0;
        }
```

```
    }

    // memcpy_async interface is bind to one queue, which will execute tasks serially
    // so memcpy-in task, compute task, memcpy-out task will keep the order
    // sync
    cnrtSyncQueue(queue);

    // free memory space
    for (int i = 0; i < inputNum; i++) {
          free(inputCpuPtrS[i]);
          cnrtFree(inputMluPtrS[i]);
    }
    for (int i = 0; i < outputNum; i++) {
          free(outputCpuPtrS[i]);
          cnrtFree(outputMluPtrS[i]);
    }
    free(inputCpuPtrS);
    free(outputCpuPtrS);
    free(param);

    cnrtDestroyQueue(queue);
    cnrtDestroyRuntimeContext(ctx);
    cnrtDestroyFunction(function);
    cnrtUnloadModel(model);
    cnrtDestroy();

    return 0;
  }
```

7.3 任务执行示例程序

下面示例展示了如何完成任务执行：

```
int inputNum, outputNum;
int64_t *inputSizeS, *outputSizeS;
cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);

// Prepare data on cpu
void **inputCpuPtrS = (void **)malloc(inputNum * sizeof(void *));
```

```
void **outputCpuPtrS = (void **)malloc(outputNum * sizeof(void *));

// Allocate I/O data memory on MLU
void **inputMluPtrS = (void **)malloc(inputNum * sizeof(void *));
void **outputMluPtrS = (void **)malloc(outputNum * sizeof(void *));

// Prepare input buffer
for (int i = 0; i < inputNum; i++) {
        inputCpuPtrS[i] = malloc(inputSizeS[i]);
        // prepare your input data here
        ...
        // malloc mlu memory
        cnrtMalloc(&(inputMluPtrS[i]), inputSizeS[i]);
        cnrtMemcpy(inputMluPtrS[i], inputCpuPtrS[i], inputSizeS[i], CNRT_MEM_TRANS_DIR_
↪HOST2DEV);
}
// Prepare output buffer
for (int i = 0; i < outputNum; i++) {
        outputCpuPtrS[i] = malloc(outputSizeS[i]);
        // malloc mlu memory
        cnrtMalloc(&(outputMluPtrS[i]), outputSizeS[i]);
}

// Prepare parameters for invoking
void **param = (void **)malloc(sizeof(void *) * (inputNum + outputNum));
for (int i = 0; i < inputNum; ++i) {
        param[i] = inputMluPtrS[i];
}
for (int i = 0; i < outputNum; ++i) {
        param[inputNum + i] = outputMluPtrS[i];
}

// Set up runtime ctx
cnrtRuntimeContext_t ctx;
cnrtCreateRuntimeContext(&ctx, function, NULL);
// Bind device
cnrtSetRuntimeContextDeviceId(ctx, 0);
cnrtInitRuntimeContext(ctx, NULL);
// Compute offline
cnrtQueue_t queue;
cnrtRuntimeContextCreateQueue(ctx, &queue);
// Invoke
```

Copyright © 2020 Cambricon Corporation.

```
cnrtInvokeRuntimeContext_V2(ctx, NULL, param, queue, (void *)&invoke_param);

cnrtSyncQueue(queue);

// copy mlu result to cpu
for (int i = 0; i < outputNum; i++) {
        cnrtMemcpy(outputCpuPtrS[i], outputMluPtrS[i], outputSizeS[i], CNRT_MEM_TRANS_DIR_
↪DEV2HOST);
}

// free memory space
for (int i = 0; i < inputNum; i++) {
        free(inputCpuPtrS[i]);
        cnrtFree(inputMluPtrS[i]);
}
for (int i = 0; i < outputNum; i++) {
        free(outputCpuPtrS[i]);
        cnrtFree(outputMluPtrS[i]);
}
free(inputCpuPtrS);
free(outputCpuPtrS);
free(param);

// free other resource
...
```

7.4 指定 DDR 通道和 Cluster 来执行 Cluster 任务

用户可以通过指定使用的 DDR 通道和访存的 DDR cluster，来提高访存效率。示例如下：

```
/* Copyright (C) [2019] by Cambricon, Inc. */
    /* resource_isolation_test */
    /*
     * A test which shows how to load and run an offline model.
     * This test consists of one operation --mlp.
     *
     */

    #include "tests.h"
```

```
int resource_isolation_test() {
  // when generating an offline model, u need cnml and cnrt both
  // when running an offline model, u need cnrt only
  cnrtInit(0);

  printf("mlp offline test\n");

  // Specify the directory where your offline file is saved.
  char fname[100] = "./cache_model/"
  // The name parameter represents the name of the offline model file.
  // It is also the name of a function in the offline model file.
  strcat(fname, name);
  // load model
  cnrtModel_t model;
  cnrtLoadModel(&model, fname);

  cnrtDev_t dev;
  cnrtGetDeviceHandle(&dev, 0);
  cnrtSetCurrentDevice(dev);

  // Bind the first channel, then malloc memmory only on first channel.
  cnrtSetCurrentChannel(CNRT_CHANNEL_TYPE_0);

  // get model total memory
  int64_t totalMem;
  cnrtGetModelMemUsed(model, &totalMem);
  printf("total memory used: %ld Bytes\n", totalMem);
  // get model parallelism
  int model_parallelism;
  cnrtQueryModelParallelism(model, &model_parallelism);
  printf("model parallelism: %d.\n", model_parallelism);

  // load extract function
  cnrtFunction_t function;
  cnrtCreateFunction(&function);
  cnrtExtractFunction(&function, model, "mlp");

  int inputNum, outputNum;
  int64_t *inputSizeS, *outputSizeS;
  cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
  cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);
```

```
    // prepare data on cpu
    void **inputCpuPtrS = (void **)malloc(inputNum * sizeof(void *));
    void **outputCpuPtrS = (void **)malloc(outputNum * sizeof(void *));

    // allocate I/O data memory on MLU
    void **inputMluPtrS = (void **)malloc(inputNum * sizeof(void *));
    void **outputMluPtrS = (void **)malloc(outputNum * sizeof(void *));

    // prepare input buffer
    for (int i = 0; i < inputNum; i++) {
          // converts data format when using new interface model
          inputCpuPtrS[i] = malloc(inputSizeS[i]);
          // malloc mlu memory
          cnrtMalloc(&(inputMluPtrS[i]), inputSizeS[i]);
          cnrtMemcpy(inputMluPtrS[i], inputCpuPtrS[i], inputSizeS[i], CNRT_MEM_TRANS_DIR_
↪HOST2DEV);
    }
    // prepare output buffer
    for (int i = 0; i < outputNum; i++) {
          outputCpuPtrS[i] = malloc(outputSizeS[i]);
          // malloc mlu memory
          cnrtMalloc(&(outputMluPtrS[i]), outputSizeS[i]);
    }

    // prepare parameters for input/output buffers
    void **param = (void **)malloc(sizeof(void *) * (inputNum + outputNum));
    for (int i = 0; i < inputNum; ++i) {
          param[i] = inputMluPtrS[i];
    }
    for (int i = 0; i < outputNum; ++i) {
          param[inputNum + i] = outputMluPtrS[i];
    }

    // setup runtime ctx
    cnrtRuntimeContext_t ctx;
    cnrtCreateRuntimeContext(&ctx, function, NULL);

    // bind device
    cnrtSetRuntimeContextDeviceId(ctx, 0);
    cnrtInitRuntimeContext(ctx, NULL);

    // compute offline
```

```
    cnrtQueue_t queue;
    cnrtRuntimeContextCreateQueue(ctx, &queue);

    // here we set affinity to first cluster
    // if u set channel, channel must corresponds to cluster
    // then we will get the maximum bandwidth utilization
    cnrtInvokeParam_t invoke_param;
    unsigned int affinity = 0x01;
    invoke_param.invoke_param_type = CNRT_INVOKE_PARAM_TYPE_0;
    invoke_param.cluster_affinity.affinity = &affinity;

    // invoke
    cnrtInvokeRuntimeContext_V2(ctx, NULL, param, queue, (void *)&invoke_param);

    // sync
    cnrtSyncQueue(queue);

    // copy mlu result to cpu
    for (int i = 0; i < outputNum; i++) {
        cnrtMemcpy(outputCpuPtrS[i], outputMluPtrS[i], outputSizeS[i], CNRT_MEM_TRANS_DIR_
↪DEV2HOST);
    }

    // free memory space
    for (int i = 0; i < inputNum; i++) {
        free(inputCpuPtrS[i]);
        cnrtFree(inputMluPtrS[i]);
    }
    for (int i = 0; i < outputNum; i++) {
        free(outputCpuPtrS[i]);
        cnrtFree(outputMluPtrS[i]);
    }
    free(inputCpuPtrS);
    free(outputCpuPtrS);
    free(param);

    cnrtDestroyQueue(queue);
    cnrtDestroyRuntimeContext(ctx);
    cnrtDestroyFunction(function);
    cnrtUnloadModel(model);
    cnrtDestroy();
```

```
    return 0;
  }
```

7.5 共享 Context 权值和指令内存示例

下面例子展示了如何在创建一个原始 Context 后，通过调用 `cnrtForkRuntimeContext()` 接口复制出 3 个能够和原始 Context 共享权值和指令内存的 Context。再将这 4 个 Context 分别传到 4 个线程来并发运行。

```
#include "cnrt.h"
#include "deprecated.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef WIN32
#include <windows.h>
#else
#include "pthread.h"
#endif
#include <unistd.h>

/*************************************************************************
 * optimal case
 *************************************************************************/
#define g_total_thread_num 4

// prepare cpu input and output
void prepareCPUInputAndOutput(int64_t *inputSizeS,
                              int inputNum,
                              void ***input,
                              int64_t *outputSizeS,
                              int outputNum,
                              void ***output) {
  void **inputCpuPtrS = (void **)malloc(sizeof(void *) * inputNum);
  void **outputCpuPtrS = (void **)malloc(sizeof(void *) * outputNum);
  for (int i = 0; i < inputNum; i++) {
    int64_t dataSize = inputSizeS[i];
    void *cpuPtr = (void *)malloc(dataSize);
    inputCpuPtrS[i] = (void *)cpuPtr;
  }
```

```
  for (int i = 0; i < outputNum; i++) {
    int64_t dataSize = outputSizeS[i];
    void *cpuPtr = (void *)malloc(dataSize);
    outputCpuPtrS[i] = (void *)cpuPtr;
  }
  *input = inputCpuPtrS;
  *output = outputCpuPtrS;
}

void writeResultToFile(int thread_id, int64_t *outputSizeS, int outputNum, void
↪**outputCpuPtrS) {
  char thread_name[2] = "0";
  thread_name[0] += thread_id;
  for (int i = 0; i < outputNum; i++) {
    int64_t outputSize = outputSizeS[i];
    int dataCount = outputSize / sizeof(float);
    char output_id[2] = "0";
    output_id[0] += i;
    char fname[100] = "mlu_output";
    strcat(fname, output_id);
    strcat(fname, "_thread");
    strcat(fname, thread_name);
    FILE *file = fopen(fname, "w");
    printf("write result to file: %s\n", fname);
    float *cpuPtr = (float *)(outputCpuPtrS[i]);
    for (int k = 0; k < dataCount; k++) {
      fprintf(file, "%f\n", cpuPtr[k]);
    }
    fclose(file);
  }
}

typedef struct threadParam {
  int64_t *inputSizeS, *outputSizeS;
  int inputNum, outputNum;
  int thread_id;
  cnrtRuntimeContext_t ctx;
} * threadParam_t;

static void *thread_func(void *args) {
  // 1. Get thread parameters.
  threadParam_t thread_param = (threadParam_t)args;
```

```
int64_t *inputSizeS = (int64_t *)thread_param->inputSizeS;
int64_t *outputSizeS = (int64_t *)thread_param->outputSizeS;
int inputNum = thread_param->inputNum;
int outputNum = thread_param->outputNum;
int thread_id = thread_param->thread_id;
cnrtRuntimeContext_t ctx = thread_param->ctx;
cnrtRet_t ret;

cnrtQueue_t queue;
cnrtRuntimeContextCreateQueue(ctx, &queue);
// 2. Prepare I/O data on CPU.
void **inputCpuPtrS;
void **outputCpuPtrS;
prepareCPUInputAndOutput(inputSizeS, inputNum, &inputCpuPtrS, outputSizeS, outputNum,
                         &outputCpuPtrS);

// 3. Prepare I/O data on MLU.
void **inputMluPtrS = (void **)malloc(sizeof(void *) * inputNum);
void **outputMluPtrS = (void **)malloc(sizeof(void *) * outputNum);

for (int i = 0; i < inputNum; i++) {
  cnrtMalloc(&(inputMluPtrS[i]), inputSizeS[i]);
  cnrtMemcpy(inputMluPtrS[i], inputCpuPtrS[i], inputSizeS[i], CNRT_MEM_TRANS_DIR_HOST2DEV);
}
for (int i = 0; i < outputNum; i++) {
  cnrtMalloc(&(outputMluPtrS[i]), outputSizeS[i]);
}
// 4. Prepare parameters for cnrtInvokeRuntimeContext.
void **param = (void **)malloc(sizeof(void *) * (inputNum + outputNum));
for (int i = 0; i < inputNum; ++i) {
  param[i] = inputMluPtrS[i];
}
for (int j = 0; j < outputNum; ++j) {
  param[inputNum + j] = outputMluPtrS[j];
}

ret = cnrtInvokeRuntimeContext(ctx, param, queue, NULL);
if (ret != CNRT_RET_SUCCESS) {
  free(param);
  goto tail;
}
```

```
  cnrtSyncQueue(queue);
  for (int i = 0; i < outputNum; i++) {
    cnrtMemcpy(outputCpuPtrS[i], outputMluPtrS[i], outputSizeS[i], CNRT_MEM_TRANS_DIR_
↪DEV2HOST);
  }
  writeResultToFile(thread_id, outputSizeS, outputNum, outputCpuPtrS);

// 5. Free thread local resources.
tail:
  cnrtDestroyQueue(queue);
  cnrtFreeArray(inputMluPtrS, inputNum);
  cnrtFreeArray(outputMluPtrS, outputNum);

  for (int i = 0; i < inputNum; i++) {
    free(inputCpuPtrS[i]);
  }
  for (int i = 0; i < outputNum; i++) {
    free(outputCpuPtrS[i]);
  }
  if (NULL != inputCpuPtrS)
    free(inputCpuPtrS);
  if (NULL != outputCpuPtrS)
    free(outputCpuPtrS);
  if (NULL != param)
    free(param);

  return NULL;
}

cnrtRet_t run(const char *name) {
  // 1. Initialize runtime library and devices.
  cnrtInit(0);

  // 2. Load offline model and extract function.
  cnrtRet_t ret;
  cnrtModel_t model;
  char fname[100] = "";
      // The name parameter represents the name of the offline model file.
      // It is also the name of a function in the offline model file.
  strcat(fname, name);
  strcat(fname, ".mef");
  printf("load file: %s\n", fname);
```

Copyright © 2020 Cambricon Corporation.

```
  cnrtLoadModel(&model, fname);
  cnrtFunction_t function;
  cnrtRuntimeContext_t rtx[g_total_thread_num] = {NULL};
  cnrtCreateFunction(&function);
  cnrtExtractFunction(&function, model, name);
  ret = cnrtCreateRuntimeContext(&rtx[0], function, NULL);
  if (ret != CNRT_RET_SUCCESS) {
    printf("Create runtime context failed!");
    return ret;
  }
  cnrtSetRuntimeContextDeviceId(rtx[0], 0);
  ret = cnrtInitRuntimeContext(rtx[0], NULL);
  if (ret != CNRT_RET_SUCCESS) {
    printf("Init runtime context failed!");
    return ret;
  }
  for (int i = 1; i < g_total_thread_num; i++) {
    cnrtForkRuntimeContext(&rtx[i], rtx[0], NULL);
  }

  // 3. Get I/O DataSize of function.
  int inputNum, outputNum;
  int64_t *inputSizeS, *outputSizeS;
  cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
  cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);

  // 4. Create thread function to run MLU in parallel.
  int thread_num = g_total_thread_num;

#ifdef WIN32
  HANDLE *tid = (HANDLE *)calloc(1, sizeof(HANDLE) * thread_num);
#else
  pthread_t *tid = (pthread_t *)calloc(1, sizeof(pthread_t) * thread_num);
#endif
  threadParam_t thread_param = (threadParam_t)calloc(1, sizeof(struct threadParam) * thread_
↪num);
  for (int i = 0; i < thread_num; ++i) {
    thread_param[i].inputSizeS = inputSizeS;
    thread_param[i].outputSizeS = outputSizeS;
    thread_param[i].inputNum = inputNum;
    thread_param[i].outputNum = outputNum;
    thread_param[i].thread_id = i;
```

```
    thread_param[i].ctx = rtx[i];
  }
  for (int i = 0; i < thread_num; ++i) {
#ifdef WIN32
    tid[i] = _beginthreadex(NULL, 0, &thread_func, thread_param + i, 0, NULL);
#else
    pthread_create(&tid[i], NULL, &thread_func, &thread_param[i]);
#endif
  }
  for (int i = 0; i < thread_num; ++i) {
#ifdef WIN32
    WaitForSingleObject(tid[i], INFINITE);
#else
    pthread_join(tid[i], NULL);
#endif
  }

  // 5. Free resources.
  cnrtDestroyFunction(function);
  for (int i = 0; i < thread_num; i++) {
    cnrtDestroyRuntimeContext(rtx[i]);
  }
  cnrtUnloadModel(model);
  cnrtDestroy();
  free(tid);
  free(thread_param);
  return CNRT_RET_SUCCESS;
}

int main(int argc, char *argv[]) {
  run("runtime_context");
  return 0;
}
```

Cambricon
寒 武 纪

8 性能调优指南

8.1 通过设置 DDR 通道和 MLU 亲和性提高访存效率

MLU270 DDR 硬件支持 4 个 DDR 通道。用户既可以设置跨 DDR 通道交织的方式访问内存，也可以指定某个 DDR 通道申请内存访问。通过调用 `cnrtSetCurrentChannel()` API 设定当前线程使用的 DDR 通道后，申请的内存就会在指定的通道中访问，否则申请的内存会通过 4 个通道交织的方式访问。

另外，用户通过设置亲和性（affinity），可以指定任务运行所使用的 cluster。如果没有设置亲和性，则由内部采用均衡调度的策略选择运行的 cluster。详情查看指定 DDR 通道和 Cluster 来执行 Cluster 任务。

设置 MLU cluster 访问所对应的 DDR 通道时，访存性能最佳。即设置 cluster0 访问 DDR 通道 0，cluster1 访问 DDR 通道 1，cluster2 访问 DDR 通道 2，cluster3 访问 DDR 通道 3。当 MLU cluster 与 DDR 通道交叉访问时，性能会有少许下降但并不明显。

在执行 Union1 任务时，用户可以设置 Union1 任务只访问某个 DDR 通道的内存，并且不同的 Union1 任务访问不同的 DDR 通道内存。另外，设置 MLU 亲和性，确保 MLU cluster 访问对应的 DDR 通道，从而彼此隔离，确保访存性能的稳定性。避免了多个不同的 Union1 任务并发访问 DDR，产生访问延时，影响性能或者带来性能波动。一般情况下，单一 Union1 任务运行时，多通道交织的方式性能优于单一通道的方式。用户需要根据实际场景适配优化，达到最优性能。

MLU220 只有一个 cluster，不区分 DDR 通道。因此该方法不适用于 MLU220。

8.2 内核调用编程模型的优化

由于内核调用是被频繁调用的计算接口，为了提升内核调用接口的性能，寒武纪对该接口内部做了优化设计。`cnrtInvokeKernel_V3()` API 在 `cnrtInvokeKernel_V2()` API 的基础上，去除冗余的初始化操作，支持亲和性功能，从而达到总体性能的提升。

详情请查看内核调用 。

8.3 提升 MLU220 数据读写性能

由于 MLU220 M.2 EDGE 的边缘侧设备 ARM CPU 与 MLU 共享 DDR，用户可以通过将 MLU 内存映射到 CPU 地址空间，使 CPU 直接访问数据，从而减少 CPU 内存到 MLU 内存拷贝的过程，从业务流程上提升

Copyright © 2020 Cambricon Corporation.

性能。

详情请查看MLU220 编程模型 。

9 CNRT 环境变量

9.1 CNRT_PRINT_INFO

功能描述

在 release 模式下，此环境变量能打印 CNRT 的 INFO 级别日志。默认不开启 INFO 级别的日志打印。

使用方法

如果开启该功能，设置该环境变量为 **true**、**on**、**yes** 和 **1** 中任意值，不区分大小写。

如果关闭该功能，设置该环境变量为 **false**、**off**、**no** 和 **0** 中任意值，不区分大小写。

9.2 CNRT_GET_HARDWARE_TIME

功能描述

在 release 模式下，运行结束后打印硬件时间（Hardware Time from Driver Event: *** Hardware time from CPU: ***）。debug 模式下默认打印。

使用方法

如果开启该功能，设置该环境变量为 **true**、**on**、**yes** 和 **1** 中任意值，不区分大小写。

如果关闭该功能，设置该环境变量为 **false**、**off**、**no** 和 **0** 中任意值，不区分大小写。

9.3 CNRT_DUMP_MLUGPR

功能描述

DUMP MLISA 程序中指定 gpr 的值到文件中。

使用方法

如果开启该功能，设置该环境变量为 **true**、**on**、**yes** 和 **1** 中任意值，不区分大小写。

如果关闭该功能，设置该环境变量为 **false**、**off**、**no** 和 **0** 中任意值，不区分大小写。

Copyright © 2020 Cambricon Corporation.

9.4 CNRT_DUMP_MLURAM

功能描述

将 MLISA 程序 nram 中指定地址的内容 DUMP 到文件中。

使用方法

如果开启该功能，设置该环境变量为 **true** 、**on** 、**yes** 和 **1** 中任意值，不区分大小写。

如果关闭该功能，设置该环境变量为 **false** 、**off** 、**no** 和 **0** 中任意值，不区分大小写。

9.5 CNRT_DUMP_MLUSCALAR

功能描述

DUMP BANG C 程序中指定标量值到文件中。

使用方法

如果开启该功能，设置该环境变量为 **true** 、**on** 、**yes** 和 **1** 中任意值，不区分大小写。

如果关闭该功能，设置该环境变量为 **false** 、**off** 、**no** 和 **0** 中任意值，不区分大小写。

9.6 CNRT_DUMP_MLUVECTOR

功能描述

将 BANG C 程序中指定向量的内容 DUMP 到文件中。

使用方法

如果开启该功能，设置该环境变量为 **true** 、**on** 、**yes** 和 **1** 中任意值，不区分大小写。

如果关闭该功能，设置该环境变量为 **false** 、**off** 、**no** 和 **0** 中任意值，不区分大小写。

9.7 CNRT_BANG_PRINTF_LIMIT

功能描述

指定 bang_printf 的缓冲区大小。单位是每个核最多 printf 记录的条数，默认值为 **1024**。

使用方法

设置此环境变量为大于零的整数值，不可以等于零。如果这个值没有设置或者不合法，就使用默认值 **1024**。

Copyright © 2020 Cambricon Corporation.

9.8 CNRT_DEFAULT_DEVICE

功能描述

如果所有线程使用同一个设备进行操作，可以设置该环境变量来指定使用的设备。该进程创建的所有子进程、所有线程将只使用该指定设备。

使用方法

设置环境变量为设备编号。环境变量数值必须为大于或等于零并且小于实际 MLU 设备数量的整型值。例如用户有 3 个 MLU 设备，那么环境变量的取值为 0、1 或 2。设置其他数值均视为非法值。设置该环境变量后，需调用 `cnrtInit()` API 完成该设备初始化。用户可以调用 `cnrtGetDeviceCount()` API 来获取设备的数目。相关 API 使用方法，请查看《寒武纪 CNRT 开发者手册》。

9.9 CNRT_BANG_PRINTF_NOT_ENABLE

功能描述

设置是否打印 BANG C 代码的调试信息，即是否启动 `BANG printf` 函数。默认值为 **false** ，即开启打印 BANG C 代码的调试信息功能。该环境变量适用于所有运行 BANG C 语言的平台。

使用方法

设置此环境变量为 **true** 即可关闭打印功能。设置为 **false** 即可开启打印功能。为了获得更好的性能，建议用户使用 debug 编译宏包上 __bang_printf 方式而不是 release 模式下的 __bang_printf 功能，并在 release 时删掉 bang printf 相关代码。更多详情，请查看《寒武纪 BANG C 开发者手册》中“BANG C Debugging”章节。

Copyright © 2020 Cambricon Corporation.