



Cambricon BANG C/C++ 编程指南

版本 1.0.0

Cambricon@155chb

2022 年 08 月 15 日



目录

目录	i
插图目录	1
表格目录	3
1 版权声明	4
2 更新历史	6
3 简介	8
3.1 Cambricon BANG 异构并行计算平台	8
3.2 Cambricon BANG 异构并行编程模型	9
3.3 文档结构	11
4 抽象硬件模型	12
4.1 存储模型	13
4.1.1 存储层次	13
4.1.1.1 GPR	15
4.1.1.2 NRAM	15
4.1.1.3 WRAM	15
4.1.1.4 SRAM	16
4.1.1.5 L2 Cache	16
4.1.1.6 LDRAM	16
4.1.1.7 GDRAM	16
4.1.2 数据迁移	17
4.1.3 访存一致性	17
4.2 计算模型	17
4.2.1 核内并行和同步	18
4.2.2 核间并行和同步	20
4.2.3 计算能力	21
5 编程模型	23
5.1 Kernel 简介	23

5.2 任务规模	25
5.3 任务类型	26
5.3.1 执行示例	27
5.4 任务映射	28
6 编程接口	31
6.1 Cambricon BANG C 编程示例	31
6.1.1 Cambricon BANG C 程序的执行流程	32
6.1.2 Cambricon BANG C 代码示例	33
6.2 编译流程	35
6.3 CNCC 常见编译选项	37
6.4 CNBin 与 CNFatbin	38
6.5 CNRT 和 CNDrv 的使用	38
6.5.1 CNRT 和 CNDrv 的区别	38
6.5.2 CNRT 的初始化	39
6.5.3 CNDrv 的初始化	40
6.5.4 设备管理接口	40
6.5.5 设备内存管理	40
6.5.5.1 设备内存拷贝	41
6.5.5.1.1 同步内存拷贝和异步内存拷贝介绍	41
6.5.5.1.2 设备内存多维同步拷贝介绍	42
6.5.5.2 云侧 MLU 平台设备内存操作介绍	48
6.5.5.2.1 MLU 加速卡平台设备 NUMA 内存管理	48
6.5.5.3 CE 边缘计算平台设备内存操作介绍	50
6.5.5.3.1 CE 边缘计算平台内存模型基础与缓存一致性	50
6.5.5.3.2 内存模型与缓存一致性	51
6.5.5.3.3 缓存一致性维护	52
6.5.5.3.4 CE 平台设备内存申请与释放	53
6.5.5.3.5 CE 平台内存映射与解映射	54
6.5.5.3.6 CE 平台缓存操作	56
6.5.5.3.7 CE 平台缓存内存使用建议	57
6.5.5.3.8 CE 平台内存信息查询	57
6.5.5.3.9 CE 平台升级指导汇总	59
6.5.6 设备 L2 Cache 管理	60
6.5.7 主机侧页锁定机制内存管理	60
6.5.8 异步并行执行	62
6.5.8.1 主机和设备之间的并发执行	62
6.5.8.2 Kernel 并发执行	63
6.5.8.3 内存拷贝和 Kernel 执行并行	63

6.5.8.4 内存拷贝任务之间并行	63
6.5.9 Queue	63
6.5.9.1 创建和销毁	63
6.5.9.2 Default Queue	65
6.5.9.3 显式同步	65
6.5.9.4 同步行为设置	65
6.5.9.4.1 同步行为的选择方法	66
6.5.9.5 不同 Queue 操作的重叠行为	67
6.5.10 Notifier	67
6.5.10.1 创建和销毁 Notifier	68
6.5.10.2 统计时间	68
6.5.10.3 多队列间同步	69
6.5.11 错误检查	70
6.6 软件版本兼容性	70
7 性能调优指南	72
7.1 性能调优的总体原则与流程	72
7.1.1 阿姆达尔定律	72
7.1.2 古斯塔夫森定律	72
7.1.3 性能调优基本原则	73
7.1.4 性能调优基本流程	73
7.2 计算效率最大化	73
7.2.1 计算并行	73
7.2.2 减少计算量	74
7.2.3 等效替代	74
7.3 IO 效率最大化	74
8 性能调优实践	75
8.1 性能分析的基本手段	75
8.1.1 CNPerf	75
8.1.1.1 CNPerf E2E 调优	75
8.1.1.1.1 寻找设备侧执行流中的热点算子	77
8.1.1.1.2 寻找设备侧执行流中的空泡	79
8.1.1.2 cnperf-cli monitor 命令	81
8.1.1.2.1 使用方法	82
8.1.1.2.2 显示效果	82
8.1.1.3 通过 cnperf-cli record --pmu 命令进行单算子调优	83
8.1.1.3.1 使用方法	83
8.1.1.3.2 显示效果	83
8.1.1.4 通过 cnperf-cli record --kernel_profiling 命令进行单算子调优	84

8.1.1.4.1 使用样例	85
8.1.1.5 小结	87
8.1.2 gettimeofday	87
8.1.3 Notifier	88
8.2 计算效率最优化	88
8.2.1 使用向量接口优化性能	88
8.2.1.1 向量相加	89
8.2.1.2 relu 激活	89
8.2.1.3 对位最大值	89
8.2.1.4 向量 split	90
8.2.1.5 矩阵乘法	91
8.2.1.6 置零操作	91
8.2.2 使用融合操作优化性能	91
8.2.3 使用查表功能实现并行间接访问	92
8.2.4 合理使用核内同步和核间同步接口	92
8.2.5 选择合适的接口实现核内指令流并行	92
8.2.6 利用多核并行计算优化性能	93
8.2.7 增加核间并行度	94
8.2.8 使用 Cluster 间流水	95
8.3 编译器常用技巧	96
8.3.1 常数预处理	96
8.3.2 使用位运算	96
8.3.3 编译 Hint	96
8.3.3.1 __builtin_assume_aligned	96
8.3.3.2 __builtin_assume	96
8.3.3.3 __builtin_unreachable	97
8.3.3.4 __restrict__	97
8.4 访存优化	98
8.4.1 减少访存数据量	98
8.4.1.1 片上驻留	98
8.4.1.2 计算换访存	99
8.4.2 计算和访存并行	99
8.4.2.1 重排代码	100
8.4.2.2 软流水	101
8.4.3 提升带宽利用率	104
8.4.3.1 数据对齐	104
8.4.3.2 张量化访存	104
8.4.3.3 阶梯式访存	105
8.5 Kernel 级别的性能优化	105

8.5.1	Kernel 融合	105
8.5.2	多队列并行	106
8.5.3	多队列流水	107
8.6	使用 Notifier 提高设备利用率	108
8.7	向量加法性能调优案例	110
8.7.1	标量版本	111
8.7.2	向量化版本	112
8.7.3	软件流水版本	113
8.7.4	多核版本	117
8.7.4.1	多核拆分方法	117
8.7.4.2	如何保证高性能	118
8.7.5	代码附录	118
9	附录	125
9.1	C++ 语言扩展	125
9.1.1	预处理符号	125
9.1.1.1	<u>--BANG_ARCH--</u>	125
9.1.1.2	<u>--MLU_NRAM_SIZE--</u>	125
9.1.1.3	<u>--MLU_SRAM_SIZE--</u>	125
9.1.1.4	<u>--MLU_WRAM_SIZE--</u>	125
9.1.2	函数修饰符	126
9.1.2.1	<u>--mlu_host--</u>	126
9.1.2.2	<u>--mlu_builtin--</u>	126
9.1.2.3	<u>--mlu_entry--</u>	126
9.1.2.4	<u>--mlu_func--</u>	126
9.1.3	地址空间修饰符	126
9.1.3.1	<u>--nram--</u>	127
9.1.3.2	<u>--wram--</u>	127
9.1.3.3	<u>--ldram--</u>	128
9.1.3.4	<u>--mlu_const--</u>	128
9.1.3.5	<u>--mlu_device--</u>	128
9.1.3.6	<u>--mlu_shared--</u>	129
9.1.4	数据类型支持	130
9.1.5	内建变量	132
9.1.5.1	<u>clusterDim</u>	132
9.1.5.2	<u>clusterId</u>	132
9.1.5.3	<u>coreDim</u>	132
9.1.5.4	<u>coreId</u>	132
9.1.5.5	<u>taskDimX</u>	132

9.1.5.6 taskDimY	132
9.1.5.7 taskDimZ	132
9.1.5.8 taskDim	133
9.1.5.9 taskIdX	133
9.1.5.10 taskIdY	133
9.1.5.11 taskIdZ	133
9.1.5.12 taskId	133
9.1.6 多核同步接口	133
9.1.6.1 __sync_cluster()	133
9.1.6.2 __sync_all_ipu()	133
9.1.6.3 __sync_all_mpu()	134
9.1.6.4 __sync_all()	134
9.1.7 地址空间识别函数	134
9.1.7.1 __is_nram()	134
9.1.7.2 __is_wram()	134
9.1.7.3 __is_sram()	134
9.1.8 格式化输出	134
9.1.9 流水线编程	136
9.1.9.1 流水线接口	136
9.1.9.2 流水线示例	137
9.2 C++ 语言标准支持	139
9.2.1 数学库函数	139
9.2.2 函数指针	140
9.2.3 动态初始化	140
9.2.4 extern 修饰符	141
9.3 可配置环境变量	141
9.3.1 CNRT_BANGC_PRINTF_LIMIT	141
9.4 CNDrv API	141
9.4.1 Context	143
9.4.1.1 Context 与设备侧资源之间的关系	144
9.4.1.2 Context 与主机侧线程之间的关系	146
9.4.2 Module	147
9.4.3 CNRT 与 CNDrv 的混合使用	148



插图目录

3.1 寒武纪软件栈架构	9
3.2 Cambricon BANG 异构并行编程模型的可扩展性	11
4.1 Cambricon BANG 异构计算平台的抽象硬件模型	13
4.2 抽象存储模型	14
4.3 多队列间数据依赖和同步	20
5.1 Cambricon BANG 异构并行编程模型	24
5.2 Task 构成的三维网格	25
5.3 物理 Cluster ID 与逻辑 Cluster ID 的映射关系	29
5.4 Cambricon BANG 异构并行编程模型示例	30
6.1 CNCC 混合编译流程	36
6.2 CNRT 的初始化流程	39
6.3 二维内存转换为一维内存	43
6.4 二维内存拷贝	45
6.5 三维内存拷贝	47
6.6 CE 边缘计算内存架构图	50
6.7 缓存一致性示意图	51
6.8 缓存一致性维护示意图	52
8.1 tracing 默认界面	76
8.2 timechart 可视化界面	76
8.3 timechart 中的设备侧执行流界面	77
8.4 选中设备侧执行流中 Kernel 的示意图	78
8.5 Wall Duration 排序显示效果	78
8.6 主机侧与设备侧执行流的整体状况	79
8.7 设备侧执行流中空泡较为明显的区域	79
8.8 操作示意图：打开 Flow event	79
8.9 计算图没有依赖的示意图	80
8.10 计算图有依赖的示意图	80
8.11 Flow event 倾斜程度示意图	81

8.12 卷积核轮转示意图	99
8.13 软流水效果图	103
8.14 CNRT 多队列流水示意	107
8.15 Notifier 流水示意	108
8.16 软件流水示意图	114
9.1 设备侧资源关系说明	144
9.2 Context 与主机侧线程关系	146

Cambricon@155chb



表格目录

4.1 不同计算能力的特性支持情况	21
5.1 任务类型为 Union2, 任务规模为 {8, 2, 2} 时内建变量的取值	28
6.1 常见的 CNCC 编译选项	37
6.2 CNRT 常用设备管理接口简介	40
6.3 新旧接口对比汇总	59
6.4 CNRT 常用 Notifier 管理接口	67
6.5 CNRT 常用错误管理接口	70
6.6 Cambricon BANG 异构计算平台版本与 CNToolkit 版本的对应关系	71
8.1 调优结果	110
9.1 Cambricon BANG C 支持的基础数据类型	131
9.2 标志字段的含义	135
9.3 长度字段的含义	135
9.4 类型字段的含义	136
9.5 Cambricon BANG C 支持的标准数学库函数和宏定义	139
9.6 CNDrv 功能模块与句柄对应关系	141



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。

Cambricon@155chb

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。

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

1. 版权声明

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。

Cambricon@155chb



2 更新历史

- **V1.0.0**

更新时间：2022 年 7 月 10 日

更新内容：

- 发布 v1.0.0 正式版本。
- 删除 VB (Video Block, 视频缓存块) 和扩展内存申请接口相关描述。
- 更新 第 8 章 性能调优实践，增加 CNPerf E2E 调优简介。

- **V0.4.11**

更新时间：2022 年 6 月 22 日

更新内容：

- 新增 第 6.4 节 CNBin 与 CNFatbin。
- 新增 第 6.5.5.3.7 节 CE 平台缓存内存使用建议。

- **V0.4.10**

Cambricon@155chb

更新时间：2022 年 5 月 05 日

更新内容：

- 更新 表 4.1 不同计算能力的特性支持情况。

- **V0.4.9**

更新时间：2022 年 4 月 20 日

更新内容：

- 更新 第 8.2.1 节 使用向量接口优化性能。

- **V0.4.8**

更新时间：2022 年 3 月 22 日

更新内容：

- 新增 第 9.1.8 节 格式化输出。
- 更新 第 8.4.1.1 节 片上驻留。
- 更新 第 6.6 节 软件版本兼容性。
- 新增 第 8.5.1 节 Kernel 融合。
- 新增 第 9.2.4 节 extern 修饰符。

- **V0.4.7**

更新时间：2022 年 3 月 05 日

更新内容：

- 新增 第 6.5.5.1 节 设备内存拷贝。

2. 更新历史

- **V0.4.6**

更新时间：2022 年 2 月 15 日

更新内容：

- 新增地址传递关系描述。
- 新增 第 9.4.3 节 CNRT 与 CNDrv 的混合使用。

- **V0.4.5**

更新时间：2021 年 12 月 23 日

更新内容：

- 修正 第 6.5.5 节 设备内存管理 描述错误。

- **V0.4.4**

更新时间：2021 年 12 月 06 日

更新内容：

- 细化 第 6.5.5 节 设备内存管理 云侧及边缘端的内存管理说明。
- 优化 图 8.14 CNRT 多队列流水示意 图文排版。

- **V0.4.3**

更新时间：2021 年 10 月 15 日

更新内容：

- 优化图片及语言描述。

- **V0.4.2**

更新时间：2021 年 09 月 30 日

更新内容：

- 新增 第 9.4 节 CNDrv API。

- **V0.4.1**

更新时间：2021 年 09 月 18 日

更新内容：

- 初始版本。



3 简介

寒武纪 MLU 是面向人工智能应用的领域专用处理器，针对人工智能领域常用的运算（例如卷积、池化和激活等）做了定制优化。与通用计算设备相比，MLU 硬件在处理人工智能应用时拥有更高的性能、能效比和灵活性。

3.1 Cambricon BANG 异构并行计算平台

为了充分发掘 MLU 硬件的计算能力，简化人工智能应用的开发，寒武纪引入了 Cambricon BANG 异构并行计算平台，并基于该平台构建了端云一体的完整解决方案，如 [图 3.1 寒武纪软件栈架构](#) 所示。

Cambricon@155chb



图 3.1: 寒武纪软件栈架构

Cambricon BANG 异构计算平台对寒武纪不同架构的硬件产品进行了高度抽象，向用户暴露了统一的编程模型和编程接口，并提供了配套的调试和分析工具。Cambricon BANG 异构编程模型用于开发各类人工智能应用程序和算子库。由于 Cambricon BANG 异构编程模型屏蔽了不同硬件的细微差异，使得基于 Cambricon BANG 异构编程模型开发的应用程序不经过修改便可以运行在包含终端、边缘侧和云端的所有寒武纪硬件上。

3.2 Cambricon BANG 异构并行编程模型

寒武纪硬件支持服务器级、板卡级、芯片级、Cluster 级、MLU Core 级、流水线级和 SIMD (Single Instruction Multiple Data，单指令多数据) 级并行。为了简化应用开发难度，充分发掘各级并行计算能力，支持并行规模的无限扩展，实现端云一体的开发、调试和部署，寒武纪推出 Cambricon BANG 异构并行编程模型。在该编程模型下，整个计算系统会被划分为设备端和主机端，二者协同完成并行计算任务。主机端用于完成对设备资源的申请和释放，并控制设备端完成任务处理，而设备端则负责大规模的

并行计算。在该编程模型下，一个完整的人工智能应用会被分解为一系列计算密集的运算核心，称为 Kernel，每个 Kernel 最终会下发到 MLU 硬件上执行。

当一个 Kernel 下发到 MLU 设备上执行时，MLU 硬件会根据主机端设置的任务规模和任务类型启动对应数量的 MLU Core 完成计算任务。一个 Kernel 会被 MLU 硬件上的一个或者多个 MLU Core 执行，每个程序实例称为一个 Task。用户不需要关心每个 Task 在哪个 MLU Core 上执行，所有的调度细节都隐藏在 Cambricon BANG 异构计算平台的内部。

Cambricon BANG 异构并行编程模型对底层硬件进行了高度抽象，使得用户不必感知底层的硬件细节，只需要遵循 Cambricon BANG 异构并行编程模型的规范，就可以实现应用程序在不同硬件平台之间的无缝迁移。

基于 Cambricon BANG 异构并行计算模型编程时，用户需要进行以下操作：

1. 任务划分：将一个计算密集的计算需求分解为多个可以并行执行的独立子任务，子任务之间可以通过共享存储的方式进行数据交互，也可以通过同步原语实现同步；
2. 任务设置：指定用于执行计算任务的硬件资源的数量。

设置任务规模和任务类型后，Cambricon BANG 异构计算平台会根据具体硬件的资源情况和设备的空闲状况执行计算任务。以 [图 3.2 Cambricon BANG 异构并行编程模型的可扩展性](#) 为例，该程序具有 8 个任务（Task）：

- 如果底层硬件有 1 个 MLU Core 可用，那么 8 个任务会以任意顺序在同一个 MLU Core 上串行执行；
- 如果底层硬件有 1 个 Cluster 可用（包含 4 个 MLU Core），那么 8 个任务会被分为 2 轮迭代，每次执行 4 个任务；
- 如果底层硬件有 2 个 Cluster 可用（包含 8 个 MLU Core），那么 8 个任务可以被同时执行。

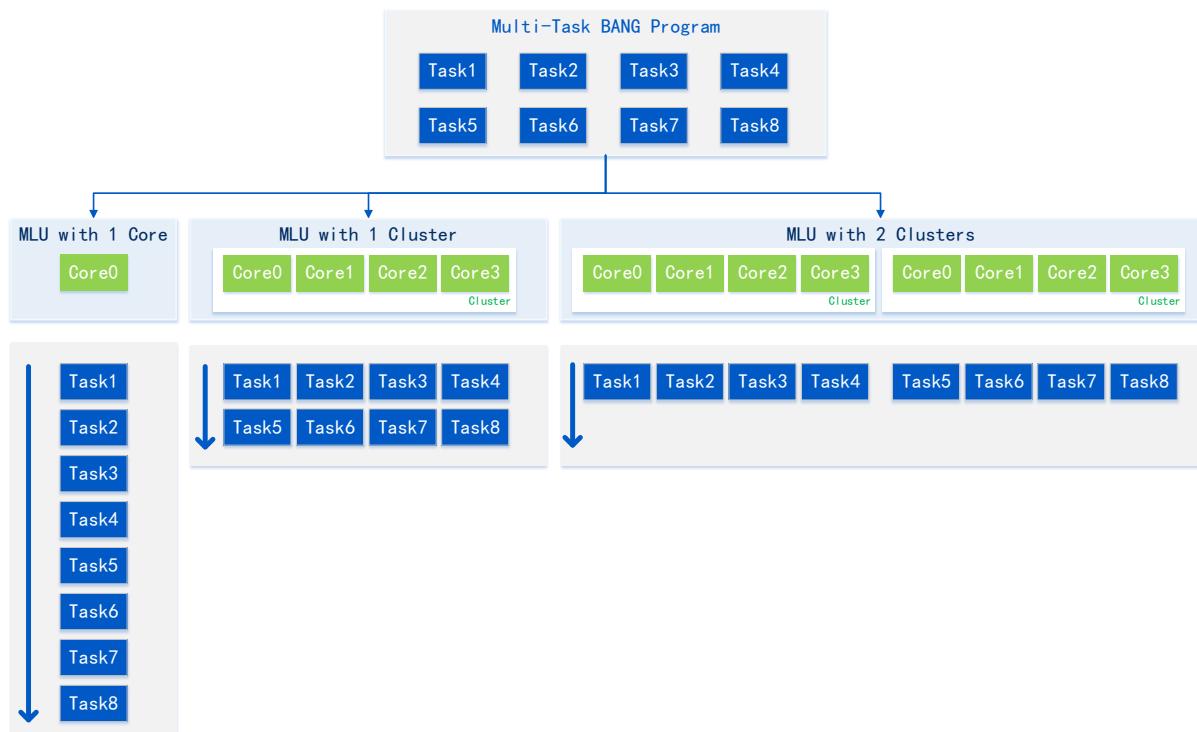


图 3.2: Cambricon BANG 异构并行编程模型的可扩展性

3.3 文档结构 Cambricon@155chb

本文档的组织结构如下：

第 4 章 抽象硬件模型 介绍寒武纪硬件的基本架构。

第 5 章 编程模型 介绍 Cambricon BANG 异构并行编程模型的基本概念。

第 6 章 编程接口 介绍 Cambricon BANG 异构并行编程模型的编程接口。

第 7 章 性能调优指南 介绍面向 Cambricon BANG 异构并行编程模型的性能设计总体策略。

第 8 章 性能调优实践 介绍面向 Cambricon BANG 异构并行编程模型的具体优化策略。



4 抽象硬件模型

寒武纪的 MLU 硬件是面向人工智能应用的领域专用处理器，针对人工智能算法的计算特性和访存特性，设计了高效的指令集、流水线、运算部件和访存部件。与通用处理器相比，MLU 硬件在处理人工智能任务时有更高的性能、灵活性和能效比。MLU 硬件针对人工智能中不同特征的访存数据流设计专用的数据通路和运算部件，实现了不同的数据流之间的隔离；同时向软件暴露了灵活的片上存储空间访问功能，提高了处理效率。

寒武纪硬件的基本组成单元是 MLU Core。每个 MLU Core 是具备完整计算、IO 和控制功能的处理器核心，可以独立完成一个计算任务，也可以与其他 MLU Core 协作完成一个计算任务。每 4 个 MLU Core 核心构成一个 Cluster，在 MLUv02 以及后续架构中，每个 Cluster 内还会包含一个额外的 Memory Core 和一块被 Memory Core 和 4 个 MLU Core 共享的 SRAM（Shared RAM，共享存储单元）。Memory Core 不能执行向量和张量计算指令，只能用于 SRAM 与 DDR（Double Data Rate Synchronous Dynamic Random Access Memory，双倍速率同步动态随机存储器，DDR SDRAM 通常简称为 DDR）和 MLU Core 之间的数据传输。

Cambricon BANG 异构并行计算平台对底层由 MLU 硬件构成的大规模并行计算系统进行了一系列抽象，屏蔽了具体硬件之间的细微差异，向用户展示了一个高度并行、灵活扩展和易于操控的抽象硬件模型。Cambricon BANG 异构并行编程模型由通用处理器和多个 MLU 领域专用处理器组成。其中，MLU 负责核心的大规模并行计算，而通用处理器则作为控制单元，负责复杂控制和任务调度等工作。整个抽象硬件模型分为 5 个层级：服务器级、板卡级、芯片级、处理器簇（Cluster）级和 MLU Core 级，每个层次都包括抽象的控制单元、计算单元和存储单元，如 [图 4.1 Cambricon BANG 异构计算平台的抽象硬件模型](#) 所示。

- 第 0 级是服务器级，由多个 CPU 构成的控制单元、本地 DDR 存储单元和多个 MLU 板卡构成的计算单元组成；
- 第 1 级是板卡级，每个 MLU 板卡由本地控制单元、DDR 存储单元和 MLU 芯片构成的计算单元组成；
- 第 2 级是芯片级，每个芯片由本地控制单元、本地存储单元（例如 L2 Cache）以及一个或者多个 Cluster 构成的计算单元组成；
- 第 3 级是 Cluster 级，每个 Cluster 由本地控制单元、共享存储以及多个 MLU Core 构成的计算单元组成；
- 第 4 级是 MLU Core 级，每个 MLU Core 由本地控制单元、私有存储单元和计算单元组成。在 MLU Core 内部支持指令级并行和数据级并行。

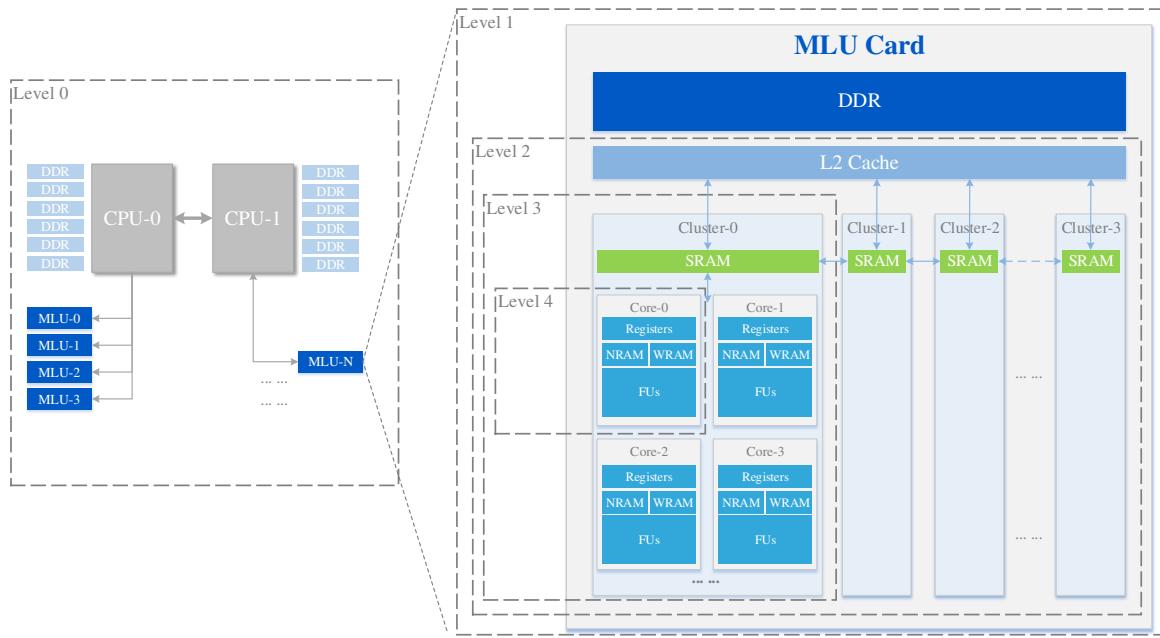


图 4.1: Cambricon BANG 异构计算平台的抽象硬件模型

注解:

- L2 Cache (Level 2 Cache, 二级缓存);
- FU (Functional Unit, 运算功能单元)。

整个抽象硬件模型可以通过增加服务器数量、板卡数量、芯片数量、Cluster 数量或者 MLU Core 数量的方式自由扩展计算能力，也可以通过在各层增加存储容量的方式自由扩展存储容量。此外，该抽象模型还可以通过增加或者减少抽象层次的方式适应不同场景的实际需求。例如，在边缘侧场景中可以省去板卡层级，直接由主机侧的 CPU 与 MLU 芯片互连；在一些计算需求不高的终端侧场景中，还可以直接省去板卡、芯片级和 Cluster 层级，将 MLU Core 作为主机侧 CPU 的协处理器。

4.1 存储模型

4.1.1 存储层次

抽象硬件模型提供了丰富的存储层次，包括 GPR (General Purpose Register, 通用寄存器)、NRAM、WRAM、SRAM、L2 Cache、LDRAM (Local DRAM, 局部 DRAM 存储单元)、GDRAM (Global DRAM, 全局 DRAM 存储空间) 等。GPR、WRAM 和 NRAM 是一个 MLU Core 的私有存储，Memory Core 没有私有的 WRAM 和 NRAM 存储资源。L2 Cache 是芯片的全局共享存储资源，目前主要用于缓存指令、Kernel 参数以及只读数据。LDRAM 是每个 MLU Core 和 Memory Core 的私有存储空间，其容量比 WRAM 和 NRAM 更大，主要用于解决片上存储空间不足的问题。GDRAM 是全局共享的存储资源，可以用于实现主机端与设备端的数据共享，以及计算任务之间的数据共享。

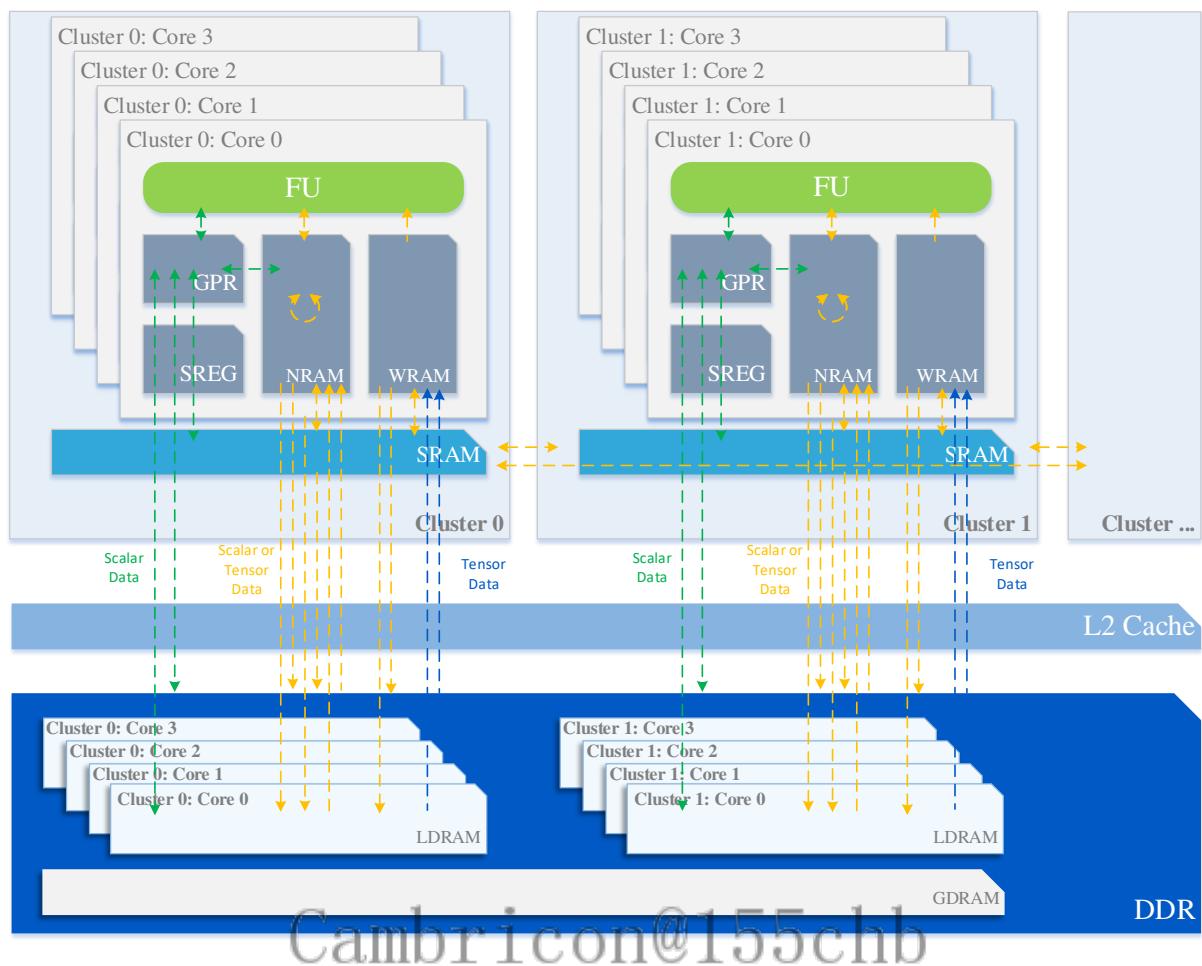


图 4.2: 抽象存储模型

注解：

SREG (Special Register, 特殊寄存器)。

注意：

- Cambricon BANG 异构计算平台的 MLU 侧采用小端字节序，即数据的高位字节存储在地址的高位。

抽象的硬件模型允许软件直接控制数据在各级存储之间的移动，从而高效地完成计算任务。为此，编译器对上层软件提供了丰富的地址空间声明，以及大量用于显式或隐式数据移动的机制和编程接口，以方便用户显式控制数据的存储空间。用户可以显式地控制数据在各个存储层次之间的移动，精确地控制数据搬运的时机和数据量，从而实现计算和 IO 之间的平衡，最大化计算效率。

4.1.1.1 GPR

GPR 是每个 MLU Core 和 Memory Core 私有的存储资源。MLU Core 和 Memory Core 的标量计算系统都采用精简指令集架构，所有的标量数据，无论是整型数据还是浮点数据，在参与运算之前必须先加载到 GPR。GPR 的最大位宽为 48 位，一个 GPR 可以存储一个 8bit、16bit、32bit 或者 48bit 的数据。GPR 中的数据不仅可以用来实现标量运算和控制流功能，还用于存储向量运算所需要的地址、长度和标量参数等。

GPR 不区分数据类型和位宽，GPR 中存储的数据的类型和位宽由操作对应数据的指令决定。当实际写入 GPR 中的数据的位宽小于 48bit 时，高位自动清零。

Cambricon BANG 异构并行编程模型中的隐式数据迁移都是借助 GPR 实现的。例如，将 GDRAM 上的标量数据赋值给一个位于 LDRAM 的变量时，编译器会自动插入访存指令将 GDRAM 中的数据先加载到 GPR 中，再插入一条访存指令将 GPR 中的数据写入 LDRAM 中。

4.1.1.2 NRAM

NRAM 是每个 MLU Core 私有的片上存储空间，主要用来存放向量运算和张量运算的输入和输出数据，也可以用于存储一些运算过程中的临时标量数据。相比 GDRAM 和 LDRAM 等片外存储空间，NRAM 有较低的访问延迟和更高的访问带宽。NRAM 的访存效率比较高但空间大小有限，而且不同硬件的 NRAM 容量不同。用户需要合理利用有限的 NRAM 存储空间，以提高程序的性能。对于频繁访问的数据，应该尽量放在 NRAM 上，仅仅当 NRAM 容量不足时，才将数据临时存储在片上的 SRAM 或者片外的 LDRAM 或者 GDRAM 上。

注意：

- 抽象硬件模型要求参与向量运算或者张量运算的输入数据和输出数据必须位于 NRAM 上。
- 位于其他存储空间的数据，在参与向量或者张量运算之前必须通过数据搬运指令显式地拷贝到 NRAM 上。

4.1.1.3 WRAM

WRAM 是每个 MLU Core 私有的片上存储空间，主要用来存放卷积运算的卷积核数据。为了高效地实现卷积运算，WRAM 上的数据具有特殊的数据布局。

提示：

WRAM 不支持普通的标量读写操作，只能通过数据搬移指令显式地读写 WRAM。

4.1.1.4 SRAM

SRAM 是一个 Cluster 内所有 MLU Core 和 Memory Core 都可以访问的共享存储空间。SRAM 可以用于缓存 MLU Core 的中间计算结果，实现 Cluster 内不同 MLU Core 或 Memory Core 之间的数据共享及不同 Cluster 之间的数据交互。

SRAM 有较高的访存带宽，但是容量有限。用户需要合理利用有限的 SRAM 存储空间，以提高程序的性能。

注意：

- SRAM 仅支持 MLUv02 及后续硬件架构。
- 由于 SRAM 是同一个 Cluster 内多个 MLU Core 共享的存储空间，每个 MLU Core 和 Memory Core 都可以自由读写 SRAM，硬件不保证所有读写操作之间的顺序，因此软件需要插入同步原语来保证数据依赖。

4.1.1.5 L2 Cache

L2 Cache 是位于片上的全局存储空间，由硬件保证一致性，目前主要用于缓存指令、Kernel 参数以及只读数据。

提示：

L2 Cache 目前对于用户透明，在 Cambricon BANG 异构并行编程模型中暂时没有提供读写 L2 Cache 的接口。

4.1.1.6 LDRAM

LDRAM 是每个 MLU Core 和 Memory Core 私有的存储空间，可以用于存储无法在片上存放的私有数据。LDRAM 属于片外存储，不同 MLU Core 和 Memory Core 之间的 LDRAM 空间互相隔离，软件可以配置其容量。与 GDRAM 相比，LDRAM 的访存性能更好，因为 LDRAM 的访存冲突比较少。

4.1.1.7 GDRAM

与 LDRAM 类似，GDRAM 也是片外存储。位于 GDRAM 中的数据被所有的 MLU Core 和 Memory Core 共享。GDRAM 空间的作用之一是用来在主机侧与设备侧传递数据，如 Kernel 的输入、输出数据等。Cambricon BANG 异构编程模型提供了专门用于在主机侧和设备侧之间进行数据拷贝的接口。

提示：

目前，GDRAM 空间只能由主机侧分配和释放，设备侧只支持 GDRAM 的读写操作。

4.1.2 数据迁移

Cambricon BANG 异构编程模型支持显式或隐式的在不同的存储层次之间实现数据迁移。

显式数据迁移由用户通过调用对应的数据移动接口完成。

隐式数据迁移由编译器自动完成，不需要用户参与。MLU 硬件要求所有的标量计算都在 GPR 中进行，当定义在 LDRAM / GDRAM / SRAM / NRAM 上的标量参与运算时，编译器会自动插入 load 指令将数据搬移到 GPR 中，在计算完成之后再通过编译器插入的 store 指令将 GPR 上的结果写回 LDRAM / GDRAM / SRAM / NRAM。

4.1.3 访存一致性

NRAM 和 WRAM 是每个 MLU Core 的私有存储空间，因此不存在多个 MLU Core 之间的访问一致性问题。但是在同一个 MLU Core 内部，属于不同流的指令读写 NRAM 或者 WRAM 时，需要软件显式插入核内同步指令以保证数据一致性；属于同一条流的指令读写 NRAM 或者 WRAM 是顺序执行的，由硬件保证一致性。

LDRAM 是每个 MLU Core 和 Memory Core 的私有存储空间，因此不存在多个 MLU Core 之间的访存一致性问题。在同一个 MLU Core 内部，读写 LDRAM 的指令顺序执行，由硬件保证一致性。

SRAM 是由多个 MLU Core 和 Memory Core 共享的存储空间，多个 MLU Core 或者 Memory Core 可以并发地读写同一个 SRAM 地址，需要由软件插入核间同步指令来保证数据一致性。

L2 Cache 是被所有 MLU Core 和 Memory Core 共享的存储空间，由硬件保证一致性。

GDRAM 是被所有 MLU Core 和 Memory Core 共享的存储空间，多个 MLU Core 或者 Memory Core 可以并发地读写同一个 GDRAM 地址，需要软件插入核间同步指令来保证数据一致性。

4.2 计算模型

MLU 硬件支持服务器级、板卡级、芯片级、Cluster 级、MLU Core 级、流水线级和数据级七个维度的并行计算。其中，服务器级和板卡级并行由具体的系统规模确定，芯片级、Cluster 级和 MLU Core 级并行由用户在主机侧配置任务规模和类型时确定，而每个 MLU Core 内部的流水线级和数据级并行计算则由用户通过对设备侧的编程来实现。在设备侧，用户编程的主体是一个 Task，每个 Task 在具体执行时只会在一个 MLU Core 上执行，而且在执行过程中不会发生任务的切换。每个 Cluster 可以并行执行多个 Task，每个芯片支持的 Cluster 数量不同。每个 MLU Core 都是具备控制流、标量运算和向量运算能力的处理器核心。标量运算指令和控制流指令主要用来实现控制流功能，而向量指令则用于实现并行数据处理。一条向量运算指令可以处理任意长度的数据。

在面向 Cambricon BANG 异构计算平台的抽象硬件模型中，每个 MLU 板卡由多块 MLU 芯片构成，每个 MLU 芯片由多个 Cluster 组成，每个 Cluster 包含多个 MLU Core、一个 Memory Core 和一块共享的

SRAM 存储单元。一个 Kernel 的所有 Task 都在同一个 MLU 硬件上执行。同一批次，在不同的 Cluster 上执行的 Task 之间可以同步，也可以通过 GDRAM 通信。

在 MLUv02 以及后续架构中，每个 Cluster 内部新增了 1 个 Memory Core。Memory Core 不具备向量和张量运算功能，只支持基本的标量运算功能。其主要功能是实现 Cluster 与 DRAM 之间、Cluster 与 Cluster 之间，以及同一个 Cluster 内部多个 MLU Core 之间的通信。

4.2.1 核内并行和同步

MLU 硬件在设计时充分考虑了人工智能应用的基本特性，设置了多个层次的并行性。MLU 硬件同时支持数据级并行和指令级并行。数据级并行是指一条指令中同时处理多个数据，向量指令就是典型的数据级并行。数据级并行的优点在于指令条数比较少。MLU 硬件同时提供了多条可以并行执行的流水线，分别对应不同的功能，位于不同流水线中的指令可以并行执行，从而实现不同流水线之间的指令级并行。

整个硬件系统由标量指令、向量指令、张量指令和访存指令构成。其中，标量系统是一个典型的 load-store 类型的 RISC (Reduced Instruction Set Computer, 精简指令集计算机) 架构。在 MLU Core 和 Memory Core 中，标量主要用来实现控制流和一些特殊的处理功能。向量指令用来实现向量运算，每条向量指令的操作数会同时包含源操作数的地址、目的操作数的地址以及向量长度。一条向量指令可以操作的数据长度是可变的，在调用对应的向量指令时，硬件会根据指令设置的向量长度进行批量处理。张量指令用于实现卷积、积分、直方图和矩阵运算。张量指令可以操作的数据长度也是可变的，在调用对应的张量指令时，硬件会根据指令设置的维度信息进行批量处理。访存指令支持变长的数据传输，可以用来实现不同存储资源之间的数据搬运，从而为标量、向量和张量运算提供数据来源。

由于向量运算指令、张量指令和访存指令都可以处理规模可变的数据，因此每一条向量指令、张量指令和访存指令的执行时间也是可变的。为了避免阻塞后续无关的指令的执行，硬件提供了多条流水线，同一条流水线中的指令串行执行，不同流水线中的指令并行执行。硬件同时提供了同步指令用于在需要维持依赖的位置实现同步。为了实现 MLU Core 内向量/张量计算和 IO 的延迟隐藏，用户应当合理安排向量/张量运算和访存指令的执行顺序。尽量通过指令调度或者重排来减少同步指令对不同流的打断，并且减少流之间的数据依赖。

MLU Core 有 4 条指令流水线，分别是 IO 流、Move 流、Compute 流和 Scalar 流。所有涉及读写片外 DDR 的指令都在 IO 流中执行，不会读写片外 DDR 的访存指令都在 Move 流中执行，张量和向量计算指令都在 Compute 流中执行，所有标量指令都在 Scalar 流中执行。所有指令流水线都是可以并行工作的，IO 流、Move 流、Compute 流和 Scalar 流默认是并行执行的，但是硬件会保证 Scalar 流与其他流之间的寄存器依赖。例如：如果有 IO 流、Move 流或者 Compute 流的指令修改标量通用寄存器时，硬件会保证 Scalar 流中读对应寄存器的指令必须在其他流的指定执行完成后才能开始执行；同理，当 Scalar 流中的指令修改寄存器时，其他流中需要读取对应寄存器的指令也需要等待 Scalar 流的写操作执行完毕才能执行。

Memory Core 可以看成是裁剪版的 MLU Core，只有 3 条指令流水线：Move 流、IO 流和 Scalar 流。不同的指令流也是可以并行工作的，由硬件保证 Scalar 流与其他流之间的寄存器依赖。

《Cambricon BANG C Developer Guide》中给出了每个 Cambricon BANG C 编程接口所属的指令流。

注解：

- 用于实现 GPR、NRAM、WRAM、SRAM 与 LDRAM、GDRAM 之间数据移动的指令，在 IO 流中执行；
- 用于实现标量运算以及控制流的指令，在 Scalar 流中执行；
- 用于实现卷积和向量运算的指令，在 Compute 流中执行；
- 用于实现 NRAM、WRAM 和 SRAM 之间数据移动的指令，在 Move 流中执行。

硬件只保证不同的指令流之间的寄存器依赖，其他依赖关系需要由用户插入核内同步指令来保证。以下指令序列为为例介绍指令流之间的依赖关系：

```
ld.gpr.gdram r0, [r0], 6;  
add.gpr.s48 r2, r0, 512;  
writezero.nram.f32 [r1], 128;  
sync;  
st.async.gdram.nram [r2], [r1], 512;  
st.async.sram.nram [r3], [r1], 512;
```

如图 4.3 多队列间数据依赖和同步 所示，MLU Core 硬件将指令队列中的上述指令序列根据指令类型发射到不同的执行队列，在没有寄存器依赖和显式同步指令的情况下，不同队列中的指令并行执行，例如本例中的“ld.gpr.gdram r0, [r0], 6”指令和“writezero.nram.f32 [r1], 128”指令。由于硬件保证了寄存器依赖，因此 Scalar 流中的“add.gpr.s48 r2, r0, 512”指令需要等待 IO 流中的“ld.gpr.gdram r0, [r0], 6”指令执行完毕后才能执行。IO 流的“st.async.gdram.nram [r2], [r1], 512”指令需要等待 Scalar 流的“add.gpr.s48 r2, r0, 512”指令执行完毕才能执行。为了保证 IO 流的“st.async.gdram.nram [r2], [r1], 512”指令和 Move 流的“st.async.sram.nram [r3], [r1], 512”指令与 Compute 流的“writezero.nram.f32 [r1], 128”指令之间的数据依赖，需要用户插入“sync”同步指令。因此，只有当 Compute 流的“writezero.nram.f32 [r1], 128”指令执行完毕以后，“st.async.gdram.nram [r2], [r1], 512”指令和“st.async.sram.nram [r3], [r1], 512”指令才能开始执行。

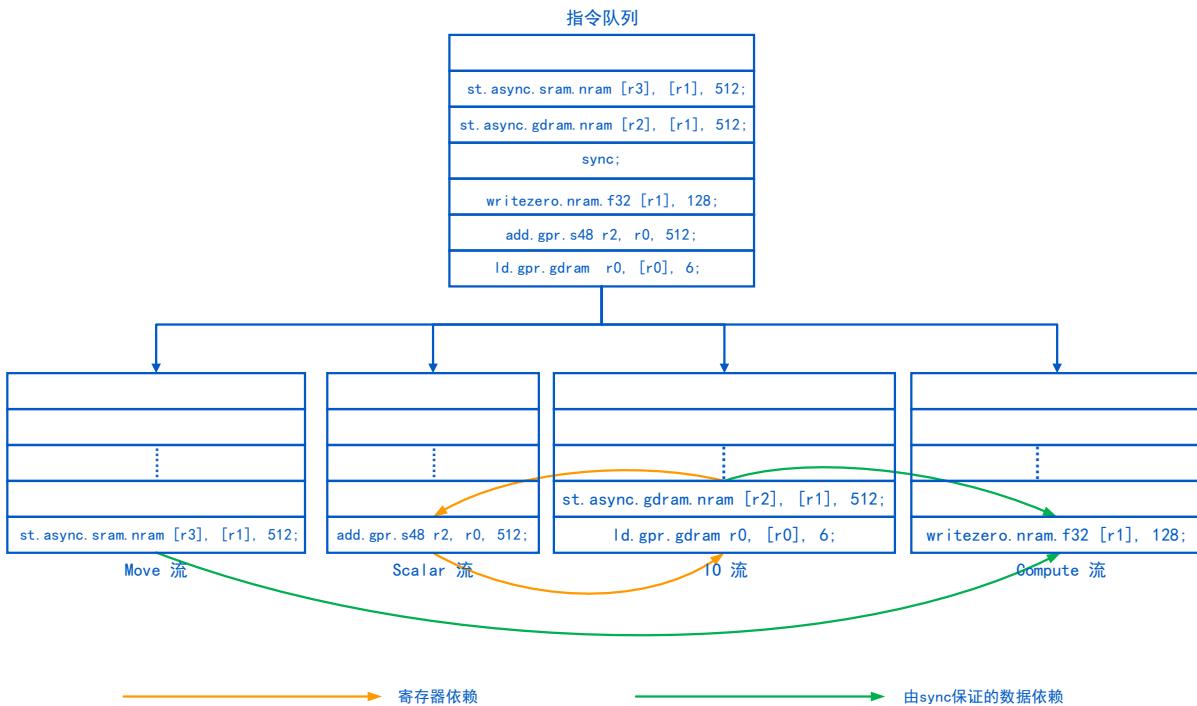


图 4.3: 多队列间数据依赖和同步

4.2.2 核间并行和同步

Cambricon@155chb

抽象硬件模型支持多个 MLU Core 协同完成同一个计算任务，一个计算任务需要的硬件资源数量由任务类型决定。对于只需要一个 Cluster 的任务，硬件会启动 4 个 MLU Core 和 1 个 Memory Core 分多轮迭代处理所有 Task。同一轮迭代的 Task 之间可以通过共享 SRAM 通信，也可以通过核间同步原语实现同步。

在抽象硬件模型中，核间同步包括局部同步和全局同步两种：

- 局部同步用于完成同一个 Cluster 内不同 MLU Core 和 Memory Core 之间的同步；
- 全局同步则用于完成多个 Cluster 之间的同步。

核间同步原语默认会同步一个核心内部的所有指令流，软件也可以指定核间同步指令只同步某些指令流。

用户应当合理划分计算任务，尽量避免使用核间同步原语。在必须使用核间同步原语时，应当控制同步范围，尽可能让最少的 MLU Core 和 Memory Core 参与同步。

为了充分利用 Cambricon BANG 异构并行平台多层次的并行计算能力，用户需要合理地进行任务划分，将任务尽量均衡地分配到尽可能多的并行计算单元上执行。对于每个层次的计算单元，通过相应的同步机制实现不同并行执行单元之间的依赖关系。例如，在主机侧通过队列同步原语实现不同 Kernel 之间的同步，在设备侧通过核间同步原语实现不同核心之间的同步，在一个计算核心内通过核内同步原语实现不同计算队列之间的同步。

用户在对 MLU 硬件编程时需要尽可能地实现计算和 IO 的并行，合理切分计算任务，用计算时间掩藏 IO 时间。从宏观上看，软件应当充分利用 Memory Core 强大的 IO 能力和 MLU Core 强大的计算能力，将

一个完整的计算任务切分为一系列子任务，子任务之间以流水线的方式依次在 Memory Core 上执行 IO 和在 MLU Core 上执行计算，MLU Core 和 Memory Core 通过 SRAM 进行数据交互。在 MLU Core 内部，还可以充分利用不同流之间的并行性，实现 Compute 流、Move 流、Scalar 流和 IO 流之间的并行。

在 [第 8 章 性能调优实践](#) 中详细介绍了利用核内、核间并行与同步实现算法性能优化的具体策略。

4.2.3 计算能力

前面介绍的硬件模型为所有 MLU 硬件的共同特性，随着硬件版本的迭代及功能的增强，不同版本的硬件存在细微的差异。为了体现这种细微差异，引入计算能力的概念，如表 4.1 不同计算能力的特性支持情况所示。

表 4.1: 不同计算能力的特性支持情况

计算能力	2.0		3.0	5.0
指令集架构	MLUv02		MLUv03	MLUv05
架构号	mtp_220	mtp_270	mtp_290	mtp_372
NRAM 容量	512KB	512KB	512KB	768KB
WRAM 容量	512KB	1024KB	512KB	1024KB
SRAM 容量	2048KB	2048KB	2048KB	4096KB
VAA 指令支持	不支持	不支持	不支持	支持
向量 float 运算	支持	支持	支持	支持
向量 half 运算	支持	支持	支持	支持
向量整型运算	不支持	不支持	不支持	支持
向量融合指令	不支持	不支持	不支持	支持
bf16 数据类型	不支持	不支持	不支持	支持
向量运算地址对齐粒度	64 字节	64 字节	64 字节	字节
向量运算长度对齐粒度	128 字节	128 字节	128 字节	sizeof(type)
三维数据搬运	不支持	不支持	不支持	支持
reduce 模式的原子操作	不支持	不支持	不支持	支持

下页继续

表 4.1 – 续上页

小卷积	不支持	不支持	不支持	支持	支持
积分功能	不支持	不支持	不支持	支持	支持
直方图功能	支持	不支持	不支持	支持	支持
超越函数功能	不支持	不支持	不支持	支持	支持
标量运算能力	单发射	单发射	单发射	双发射	双发射
离散原子操作	不支持	不支持	不支持	不支持	支持
离散向量拷贝	不支持	不支持	不支持	不支持	支持
Cache Mode	不支持	不支持	不支持	不支持	支持
向量 tf32 运算支持	不支持	不支持	不支持	不支持	支持
Cluster 间直接通信	支持	支持	支持	支持	不支持

Cambricon@155chb



5 编程模型

本节主要介绍 Cambricon BANG 异构并行编程模型的基本概念，完整的 Cambricon BANG C/C++ 代码示例可以参考[Cambricon BANG C 编程示例](#)。

5.1 Kernel 简介

Cambricon BANG 异构并行编程模型利用 CPU 和 MLU 协同计算，实现了 CPU 和 MLU 的优势互补。在 Cambricon BANG 异构并行编程模型中，CPU 作为主机侧的控制设备，用于完成复杂的控制和任务调度；而设备侧的 MLU 则用于大规模并行计算和领域相关的计算任务。

在 Cambricon BANG 异构并行编程模型中，在 MLU 上执行的程序称作 Kernel。每个 Task 都执行一次对应的 Kernel 函数。在 MLU 上可以同时执行多个并行的 Kernel。

在 Cambricon BANG C 语言中，设备侧的 Kernel 是一个带有 `__mlu_entry__` 属性的函数，该函数描述一个 Task 需要执行的所有操作。在 Kernel 内部还可以通过 `taskId` 等内建变量获得每个 Task 唯一的 ID，从而实现不同 Task 的差异化处理。此外，类似的内建变量还包括 `clusterId`、`taskIdX`、`taskIdY` 等。

以下示例定义了一个用于实现向量加法的 Kernel 函数，该示例将一个完整的向量加法操作拆分成多个可以独立的任务，每个任务实现 64 个元素的加法。

```
#define SIZE_PER_TASK 64

__mlu_entry__ void Kernel(half* dst, half* src1, half* src2) {
    __nram__ half output[SIZE_PER_TASK];
    __nram__ half input1[SIZE_PER_TASK];
    __nram__ half input2[SIZE_PER_TASK];
    __memcpy(input1, src1 + SIZE_PER_TASK * taskId, SIZE_PER_TASK * sizeof(half), GDRAM2NRAM);
    __memcpy(input2, src2 + SIZE_PER_TASK * taskId, SIZE_PER_TASK * sizeof(half), GDRAM2NRAM);
    __bang_add(output, input1, input2, SIZE_PER_TASK);
    __memcpy(dst + SIZE_PER_TASK * taskId, output, SIZE_PER_TASK * sizeof(half), NRAM2GDRAM);
}
```

Cambricon BANG C 语言提供了语法糖 <<<...>>> 用于在主机侧以类似普通函数调用的方式启动一个 Kernel：

```
#include "bang.h"

int main() {
    ...
    Kernel<<<dim, ktype, pQueue>>>(mlu_result, mlu_source1, mlu_source2);
    ...
}
```

其中，`<<<dim, ktype, pQueue>>>` 中的 `dim` 表示任务规模，详细描述请参见 第 5.2 章 任务规模；`pQueue` 表示该 Kernel 将会放到哪个任务队列中执行，详细描述请参见 第 6 章 编程接口；`ktype` 表示任务类型，即 Kernel 执行需要的硬件资源数量，详细描述请参见 第 5.3 章 任务类型。

在主机侧使用 `<<<dim, ktype, pQueue>>>` 语法糖启动的 Kernel 会异步执行，主机侧不需要等待 Kernel 执行完毕即可继续执行后续的代码。Cambricon BANG 异构并行计算平台会将对应的 Kernel 插入对应的执行队列中，并在设备侧有资源空闲时调度 Kernel 到硬件上执行。

如 图 5.1 Cambricon BANG 异构并行编程模型 所示，主机侧顺序启动 Kernel1、Kernel2 和 Kernel3。具体执行顺序如下：

- Kernel1 和 Kernel3 位于同一个执行队列中，因此 Kernel3 需要等待 Kernel1 执行完毕才能开始执行；
- Kernel2 与 Kernel1 和 Kernel3 不在同一个队列中，因此 Kernel2 可以与 Kernel1 或 Kernel3 并行执行。

Cambricon@155chb

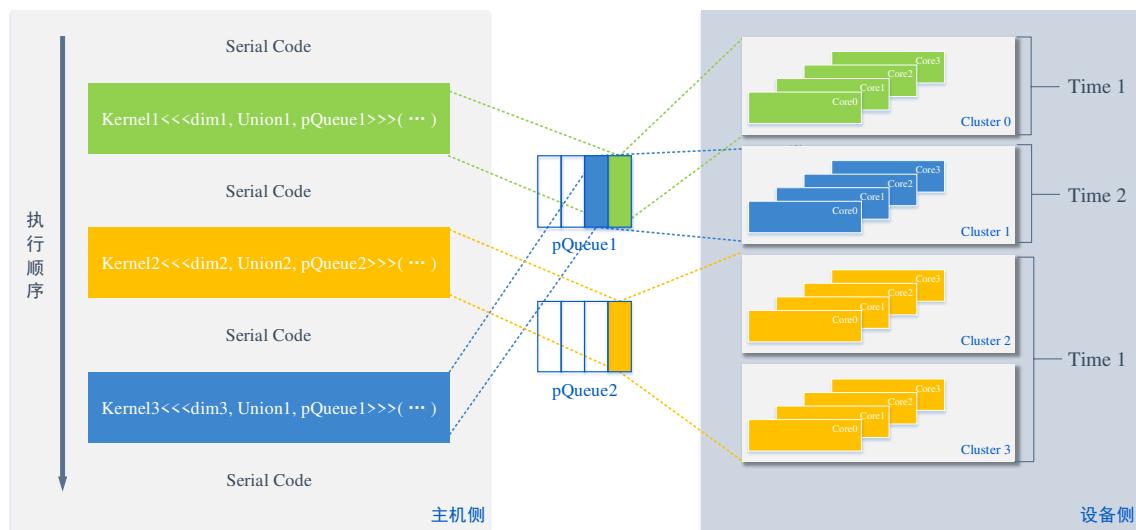


图 5.1: Cambricon BANG 异构并行编程模型

5.2 任务规模

在 Cambricon BANG 异构并行编程模型中，一个 Kernel 描述了一个 Task 的行为。在具体编程过程中，用户需要将一个完整的计算任务拆解为一系列可以并行的 Task，所有的 Task 构成一个三维网格。这个三维网络的维度信息由用户做任务拆分时确定。在由 Task 构成的三维网格中，每个 Task 都有唯一的坐标。每个任务除了一个三维坐标以外，还有一个全局唯一的线性 ID。在实际执行时，每个 Task 会映射到一个物理 MLU Core 上执行。MLU Core 在执行一个 Task 的过程中不会发生切换，只有一个 Task 执行完毕，另一个 Task 才能开始执行。

为了便于描述任务规模，在 Cambricon BANG C 编程语言中引入了 `cnrtDim3_t` 数据类型：

```
cnrtDim3_t dim;  
dim.x = 8;  
dim.y = 8;  
dim.z = 4;
```

上述配置描述的三维任务网格如 图 5.2 Task 构成的三维网格 所示。

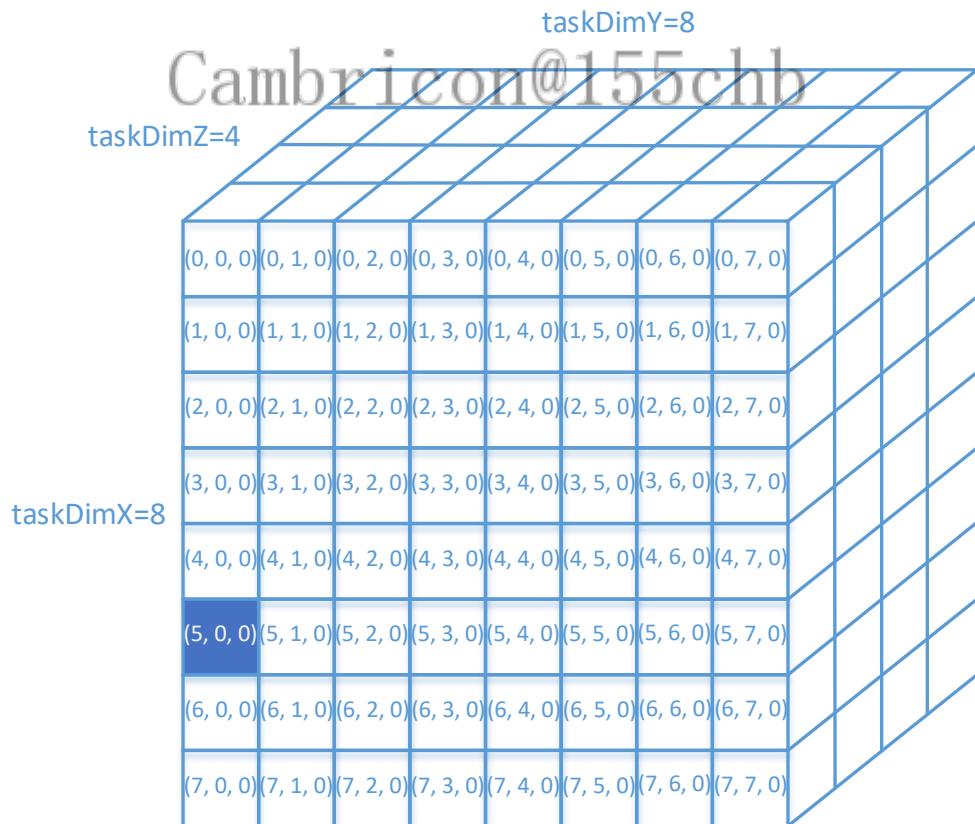


图 5.2: Task 构成的三维网格

注解：

比如，图 5.2 Task 构成的三维网格 中蓝色任务的三维坐标为 (5, 0, 0) , 线性编号为 5。

Cambricon BANG C 语言为用户提供了一系列内置变量来显式并行编程。其中，与任务规模相关的内置变量包括：taskDim、taskDimX、taskDimY 以及 taskDimZ。

- taskDimX、taskDimY 和 taskDimZ 分别对应任务规模的三个维度：dim.x、dim.y 和 dim.z。
- taskIdX、taskIdY 和 taskIdZ 的取值范围分别是 [0, taskDimX-1] , [0, taskDimY-1] , [0, taskDimZ-1] 。
- taskDim 等于 taskDimX、taskDimY 和 taskDimZ 三者的乘积。

在 图 5.2 Task 构成的三维网格 所示的任务规模配置为 taskDimX=8, taskDimY=8, taskDimZ=4, taskDim = 256 。

对于每个任务，可以通过内置变量 taskIdX、taskIdY 和 taskIdZ 获得本任务在 X 方向、Y 方向和 Z 方向的坐标。也可以通过 taskId 获得本任务在整个三维任务块中的线性编号，即 taskId = taskIdZ * taskDimY * taskDimX + taskIdY * taskDimX + taskIdX。taskId 的取值范围是 [0, taskDim - 1] 。

基于 图 5.2 Task 构成的三维网格 所示的任务规模配置，如果要实现两个长度为 16384 的向量加法，那么每个任务需要处理 $16384 / 8 / 8 / 4 = 64$ 个元素。使用 Cambricon BANG C 实现的 Kernel 函数如下所示：

```
#define N 64

__mlu_entry__ add(float* x, float* y, float* z) {
    __nram__ float x_tmp[N];
    __nram__ float y_tmp[N];
    __memcpy(x_tmp, x + taskId * N, N * sizeof(float), GDRAM2NRAM);
    __memcpy(y_tmp, y + taskId * N, N * sizeof(float), GDRAM2NRAM);
    __bang_add(x_tmp, x_tmp, y_tmp, N);
    __memcpy(z + taskId * N, x_tmp, N * sizeof(float), NRAM2GDRAM);
}
```

5.3 任务类型

任务类型指定了一个 Kernel 所需要的硬件资源数量，即一个 Kernel 在实际执行时会启动多少个物理 MLU Core 或者 Cluster。在 Cambricon BANG 异构并行编程模型中支持两种任务类型：Block 任务和 Union 任务。

Block 任务代表一个 Kernel 在执行时至少需要占用一个 MLU Core。对于 Block 类型的任务，不支持共享 SRAM，不支持不同 Cluster 之间的通信。Block 任务是所有寒武纪硬件都支持的任务类型。当任务规模大于 1 时，由 Cambricon BANG 异构并行计算平台根据硬件资源占用情况决定所有任务占用的 MLU

Core 数量：如果只有一个 MLU Core 可用，那么所有任务在同一个 MLU Core 上串行执行；如果有多个物理 MLU Core 可用，那么所有任务会被平均分配到所有可用的 MLU Core 上分批次执行。

UnionN ($N=1, 2, 4, 8, \dots$) 任务表示一个 Kernel 在执行时至少需要占用 N 个 Cluster，其中，N 必须为 2 的整数次幂。一个拥有 M 个 Cluster 的硬件，N 的最大值为 $2^{\lfloor \log_2 M \rfloor}$ 。MLU 硬件对 Union 任务的支持与硬件的具体配置有关。例如，一些终端侧或者边缘侧的单核设备不支持 Union 任务，而一个拥有 8 个 Cluster 的硬件只能够支持 Union1、Union2、Union4 和 Union8 类型的 Union 任务，无法支持 Union16 类型的任务。

注意：

- 对于 Union1 类型的任务，`dim.x` 必须是 4 的倍数，支持一个 Cluster 中的 4 个 MLU Core 和一个 Memory Core 共享 SRAM，但是不支持 Cluster 之间的通信。
- 对于 Union2/4/8/16 类型的任务，`dim.x` 必须是 8/16/32/64 的倍数，而且每个 Cluster 中的 4 个 MLU Core 和 Memory Core 共享 SRAM，支持 2/4/6/8 个 Cluster 之间的通信。
- Cambricon BANG 异构并行编程模型不支持 `taskIdY` 和 `taskIdZ` 不同的任务之间通过共享 SRAM 通信。

硬件当前空闲的物理 Cluster 数量大于 UnionN 任务类型所需要的 Cluster 数量时，由 Cambricon BANG 异构并行计算平台根据硬件资源的占用情况决定是否将任务平均分配到更多的 Cluster 上执行。

5.3.1 执行示例 Cambricon@155chb

下面通过一个例子来阐明当程序运行时并行内置变量的值。

注解：

示例说明：

- 任务类型为 Union2；
- 任务规模为 {x=8, y=2, z=2}；
- `clusterDim` 对应任务类型，在本例中 `clusterDim = 2`；
- `taskDimX`, `taskDimY`, `taskDimZ` 分别对应任务规模，在本示例中，`taskDimX=8`, `taskDimY=2`, `taskDimZ=2`, `taskDim = 8 * 2 * 2 = 32`。

`taskIdX`, `taskIdY`, `taskIdZ` 和 `taskId` 的值如下表所示：

表 5.1: 任务类型为 Union2, 任务规模为 {8, 2, 2} 时内建变量的取值

Core	taskId				taskIdX				taskIdY				taskIdZ				clusterId				coreId			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	8	16	24	0				0	1	0	1	0		1	0					0			
1	1	9	17	25	1																1			
2	2	10	18	26	2																2			
3	3	11	19	27	3																3			
4	4	12	20	28	4																0			
5	5	13	21	29	5																1			
6	6	14	22	30	6																2			
7	7	15	23	31	7																3			

注解：

本示例中，

Cambricon@155chb

- `taskDimX = 8`，因此需要同时占用 8 个 MLU Core，分别对应上面表格的 8 行；
 - 整个任务需要 4 轮迭代才能执行完毕，每个内建变量在每一轮的取值如上述表格的各列所示。

5.4 任务映射

Cambricon BANG 异构并行编程模型借助运行时环境实现任务映射和任务调度，即确定某个任务具体在哪个设备或者计算单元上以何种顺序执行。在 Cambricon BANG 异构并行编程模型中，控制逻辑和串行任务多在主机端运行，计算部分和并行任务多在设备端执行。运行时提供了上述异构执行流程中主机端的编程接口，便于主机端调用以启动设备。主机端运行时接口可以通过任务队列来管理要执行的任务。运行时不断地把任务放到任务队列中，一旦硬件资源空闲，运行时就从任务队列中取一个任务出来执行。设备端的运行时调度可以由软件来完成，也可以由硬件完成，目标是保证在不同架构的处理器上能够充分地利用硬件资源。

任务类型描述的是任务的硬件需求，任务规模描述的是任务的划分方式，而任务映射则建立了具体任务与物理硬件之间的映射关系。

MLU 任务映射时需要满足以下基本原则：

1. 对于 Block 任务，只要有一个 MLU Core 空闲，即可以将任务下发。Block 任务的最小的并行度为 1。在不考虑任务展开（即将所有任务平均分配到多个 MLU Core 上并行执行）的情况下，迭代次

- 数 = $\text{taskDimZ} * \text{taskDimY} * \text{taskDimX}$ 。驱动在下发任务时会根据当前硬件的利用率决定是否将 Block 任务展开，尽可能地占满所有的硬件资源。
2. 对于 UnionN 类型的任务，映射后的起始物理 Cluster ID 要满足对齐约束，即起始物理 Cluster ID 必须能够被 N 整除。因此，对于 6 个 Cluster 的硬件，如果先下发了一个 Union2 任务，那么必须等 Union2 任务完成以后，才能下发 Union4 类型的任务。
 3. 对于 UnionN 类型的任务，需要至少有 N 个 Cluster 空闲时才能下发任务。同时还要满足约束 $\text{taskDimX \% (N * coreDim)} = 0$ 。最小的并行度为 $\text{clusterDim} * \text{coreDim}$ 。在不考虑任务展开的情况下，迭代次数 = $\text{taskDimZ} * \text{taskDimY} * \text{taskDimX} / (\text{N} * \text{coreDim})$ 。驱动根据当前硬件的实时利用率决定是否将任务展开，尽可能占满所有的硬件资源。例如：当任务类型为 Union1，任务规模为 {X=8, Y=1, Z=1} 时，默认情况下，该任务会在一个 Cluster 上经过两轮迭代才成完成。如果，驱动在下发任务时发现硬件至少有两个 Cluster 空闲，那么还会做任务展开，让 Union1 类型的任务同时占用 2 个 Cluster，这样只需要一轮迭代即可完成。
 4. 用户可以通过任务类型来控制 Union 任务的最小并行粒度，但是必须确保 taskDimX 是 $\text{clusterDim} * \text{coreDim}$ 的正整数倍。

接下来，以 [图 5.3 物理 Cluster ID 与逻辑 Cluster ID 的映射关系](#) 所示演示前述的任务映射规则。假设软件侧依次下发了 4 个 Union 任务：

- 第一个 Union1 类型的任务只需要占用一个物理 Cluster，本例中映射到了硬件的第一个 Cluster 上；
- 第二个 Union2 类型的任务需要至少占用 2 个物理 Cluster，考虑到对齐约束，本例中映射到了第三个 Cluster 开始的 2 个连续的物理 Cluster 上；
- 第三个 Union4 类型的任务映射到了从第 5 个 Cluster 开始的连续 4 个物理 Cluster 上；
- 第四个 Union4 类型的任务被展开后运行在了第 9 个物理 Cluster 开始的连续的 8 个 Cluster 上。

注意：

物理 Cluster ID 是整个芯片全局唯一的编号，而逻辑 Cluster ID 则是每个任务内部独立编码的。

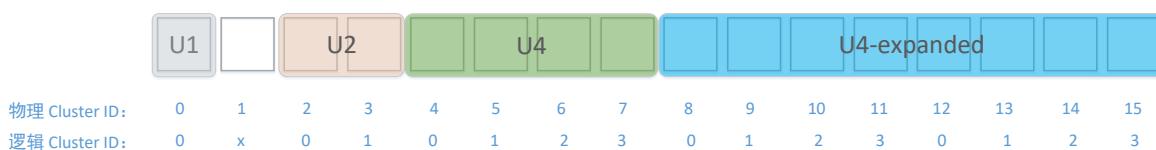


图 5.3: 物理 Cluster ID 与逻辑 Cluster ID 的映射关系

最后，以多个 Kernel 并行执行为例，如 [图 5.4 Cambricon BANG 异构并行编程模型示例](#) 所示，在主机侧启动了三个没有任何依赖的 Kernel：

- Kernel1 的任务类型为 Union2，任务规模为 {8, 1, 1}；
- Kernel2 的任务类型为 Union1，任务规模为 {8, 1, 1}；
- Kernel3 的任务类型为 Union4，任务规模为 {16, 2, 1}。

那么在一个只有 4 个 Cluster 的 MLU 硬件上执行顺序如下：

1. 主机侧先将 Kernel1 的任务类型和任务规模传递给设备侧的任务调度器。任务调度器会在设备侧有两

- 个满足对齐约束的连续的物理 Cluster 空闲时才启动 Kernel1。本例中，Kernel1 最终在 Cluster 0 和 Cluster 1 上执行, 共计占用 8 个 MLU Core, taskDimX 刚好为 8, 因此只需要一轮迭代即可执行完毕;
2. 由于 Kernel2 和 Kernel1 没有任何依赖关系, 设备侧的调度器会在硬件有一个 Cluster 空闲时开始执行 Kernel2, 不需要等待 Kernel1 执行完毕。本例中, Kernel2 最终在 Cluster 2 上执行, 共计占用 4 个 MLU Core, 而 TaskDimX 为 8, 因此需要两轮迭代即可执行完毕; Kernel1 与 Kernel2 分别在不同的 Cluster 上并行执行;
 3. 由于 Kernel3 和 Kernel1、Kernel2 都没有依赖关系, 因此设备侧会在硬件有 4 个满足对齐约束的连续的硬件 Cluster 空闲时才能执行 Kernel3。本例中, Kernel3 需要等待 Kernel1 和 Kernel2 执行完毕, 才能获得足够的硬件资源; 由于 TaskDimY 为 2, 因此需要 2 轮迭代才能执行完毕。

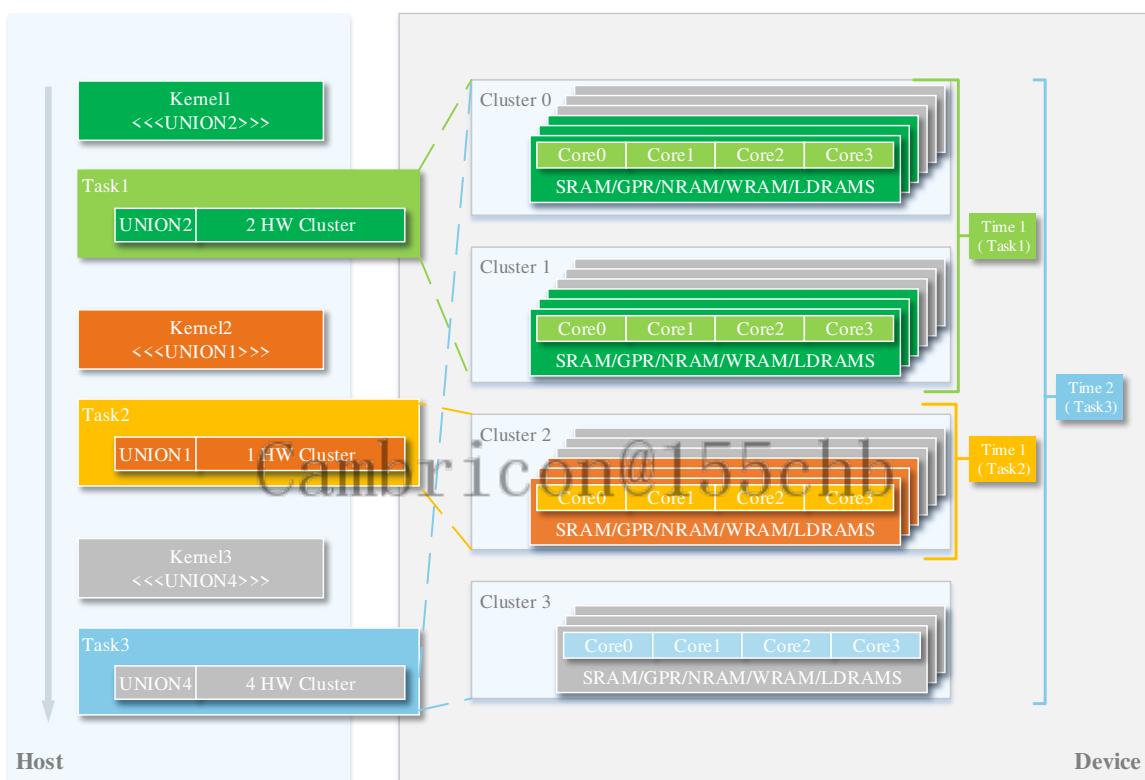


图 5.4: Cambricon BANG 异构并行编程模型示例



6 编程接口

Cambricon BANG 异构计算平台的核心组件是面向 MLU 硬件的编译器工具链，目前支持通过 C/C++ 的扩展语言 Cambricon BANG C 和基于 Python 的扩展语言 Cambricon BANGPy 对 MLU 硬件进行编程。编译器工具链和编程语言为任务划分、并行处理、数据通信和同步机制等提供底层支持，使用户可以专注于编写应用的处理逻辑本身。利用 Cambricon BANG C 和 Cambricon BANGPy 编程语言可以开发各类人工智能应用和算法库，并最终形成完整的人工智能解决方案。

Cambricon BANG C 在 C/C++ 语言的基础上，增加了 Cambricon BANG 异构并行编程模型必须的语法特性、计算原语、数据类型和内建变量支持，具体的语言扩展参考附录[C++ 语言扩展](#)。此外，Cambricon BANG C 针对异构编程环境的特点对 C/C++ 进行了简化，禁用了一些不适合异构编程环境的 C/C++ 特性，具体的语言限制见附录[C++ 语言标准支持](#)。Cambricon BANG C 程序可以使用 CNGDB 进行调试，有关 CNGDB 的使用方法可以参考《寒武纪 CNGDB 用户手册》。

Cambricon BANG C 语言整合了不同类型指令集和架构的计算单元，并支持寒武纪推出的云端、边缘端和终端设备。遵循 Cambricon BANG C 编程规范的应用程序，几乎可以无需修改直接运行在包含不同 MLU Core 数量、Cluster 数量的 MLU 硬件上。使用 Cambricon BANG C 编写的异构程序包括主机侧和设备侧的代码。其中，主机侧主要是借用 CNRT (Cambricon Runtime Library，寒武纪运行时库) 或者 CNDrv (Cambricon Driver API，寒武纪软件栈驱动接口) 提供的相关接口实现设备信息查询、设备选择、设备内存分配、任务队列创建、输入数据或参数准备、任务描述、Kernel 启动、输出获取等功能；而设备侧的入口函数就是 Kernel 函数，Kernel 函数中可以使用 Cambricon BANG C 面向设备侧编程时扩展的语法特性、计算原语、数据类型和内建变量等特性。

本章先以一个简单的向量加法为例介绍 Cambricon BANG C 编程的基本流程，然后再分别介绍 Cambricon BANG C 程序的编译流程以及 CNRT 和 CNDrv 相关的接口。

6.1 Cambricon BANG C 编程示例

使用 Cambricon BANG C 编写程序时，需要同时编写主机侧和设备侧的代码。其中，主机侧程序通过调用 CNRT 或者 CNDrv 接口完成设备初始化、设备内存管理、主机端与设备端的数据拷贝、启动 Kernel、释放设备资源等工作；设备端程序由多个 Kernel 构成。不同的 Kernel 之间可以并行执行，也可以串行执行，用户可以调用 CNRT 或者 CNDrv 接口进行控制。

6.1.1 Cambicon BANG C 程序的执行流程

典型的 Cambicon BANG C 程序执行流程如下：

1. 通过 CNRT 接口选择硬件设备：

```
unsigned int count = 0;
cnrtGetDeviceCount(&count);
cnrtSetDevice(0);
```

2. 在主机侧准备输入数据，并为输出数据分配空间。
3. 在主机侧调用 CNRT 接口分配设备内存，并将输入数据拷贝到设备内存。

```
half* mlu_input;
cnrtMalloc((void**)(&mlu_input), dims_a * sizeof(half));
cnrtMemcpy(mlu_input, input_half, dims_a * sizeof(half), cnrtMemcpyHostToDev);
```

4. 设置 Kernel 的任务规模：CNRT 定义了 `cnrtDim3_t` 数据类型用来设置 Kernel 的任务规模：

```
cnrtDim3_t dim;
dim.x = 1;
dim.y = 1;
dim.z = 1;
```

5. 设置 Kernel 的任务类型：CNRT 定义了 `cnrtFunctionType_t` 数据类型用来设置任务类型，其值可以是 Block 或 UnionN (N = 1, 2, 4, 8)。

```
cnrtFunctionType_t ktype = cnrtFuncTypeBlock;
```

6. 通过 CNRT 接口创建任务队列：

```
cnrtQueue_t pQueue;
cnrtQueueCreate(&pQueue);
```

7. 向任务队列添加 Kernel：

```
Kernel<<<dim, ktype, pQueue>>>(...);
```

8. 调用 CNRT 接口等待任务队列执行完成：

```
cnrtQueueSync(pQueue);
```

9. 调用 CNRT 接口将计算结果拷贝至主机侧：

```
cnrtMemcpy(output_half, mlu_output, sizeof(half), cnrtMemcpyDevToHost);
cnrtCastDataType_V2(&output_half[0], cnrtHalf, output, cnrtFloat, 1, nullptr, cnrtRound_rm);
```

10. 释放主机侧和设备侧的各类资源。这些资源主要包括任务队列、设备侧内存、主机侧内存等。

```
cnrtQueueDestroy(pQueue);
cnrtFree(mlu_input);
free(output_half);
```

6.1.2 Cambicon BANG C 代码示例

Cambicon BANG C 程序的文件后缀是 *.mlu，头文件的后缀和 C/C++ 语言一样，例如 *.h。Cambicon BANG C 异构程序必须包含头文件 bang.h，该头文件包含了混合编程必需的数据类型的定义以及函数接口声明。

接下来，以一个简单的向量加法为例演示 Cambicon BANG C 编写异构并行程序的基本方法。

```
#include <bang.h>

#define EPS 1e-7
#define LEN 1024

__mlu_entry__ void Kernel(float* dst, float* source1, float* source2) {
    __nram__ float dest[LEN];
    __nram__ float src1[LEN];
    __nram__ float src2[LEN];
    __memcpy(src1, source1, LEN * sizeof(float), GDRAM2NRAM);
    __memcpy(src2, source2, LEN * sizeof(float), GDRAM2NRAM);
    __bang_add(dest, src1, src2, LEN);
    __memcpy(dst, dest, LEN * sizeof(float), NRAM2GDRAM);
}

int main(void)
{
    cnrtQueue_t queue;
    CNRT_CHECK(cnrtSetDevice(0));
    CNRT_CHECK(cnrtQueueCreate(&queue));

    cnrtDim3_t dim = {1, 1, 1};
    cnrtFunctionType_t ktype = CNRT_FUNC_TYPE_BLOCK;

    cnrtNotifier_t start, end;
    CNRT_CHECK(cnrtNotifierCreate(&start));
    CNRT_CHECK(cnrtNotifierCreate(&end));

    float* host_dst = (float*)malloc(LEN * sizeof(float));
```

```
float* host_src1 = (float*)malloc(LEN * sizeof(float));
float* host_src2 = (float*)malloc(LEN * sizeof(float));

for (int i = 0; i < LEN; i++) {
    host_src1[i] = i;
    host_src2[i] = i;
}

float* mlu_dst;
float* mlu_src1;
float* mlu_src2;

CNRT_CHECK(cnrtMalloc((void**)&mlu_dst, LEN * sizeof(float)));
CNRT_CHECK(cnrtMalloc((void**)&mlu_src1, LEN * sizeof(float)));
CNRT_CHECK(cnrtMalloc((void**)&mlu_src2, LEN * sizeof(float)));

CNRT_CHECK(cnrtMemcpy(mlu_src1, host_src1, LEN * sizeof(float), cnrtMemcpyHostToDev));
CNRT_CHECK(cnrtMemcpy(mlu_src2, host_src2, LEN * sizeof(float), cnrtMemcpyHostToDev));

CNRT_CHECK(cnrtPlaceNotifier(start, queue));
Kernel<<<dim, ktype, queue>>>(mlu_dst, mlu_src1, mlu_src2);
CNRT_CHECK(cnrtPlaceNotifier(end, queue));

cnrtQueueSync(queue);
CNRT_CHECK(cnrtMemcpy(host_dst, mlu_dst, LEN * sizeof(float), cnrtMemcpyDevToHost));

for (int i = 0; i < LEN; i++) {
    if (fabsf(host_dst[i] - 2 * i) > EPS) {
        printf("%f expected, but %f got!\n", (float)(2 * i), host_dst[i]);
    }
}

float timeTotal;
CNRT_CHECK(cnrtNotifierDuration(start, end, &timeTotal));
printf("Total Time: %.3f ms\n", timeTotal / 1000.0);

CNRT_CHECK(cnrtQueueDestroy(queue));

cnrtFree(mlu_dst);
cnrtFree(mlu_src1);
cnrtFree(mlu_src2);
free(host_dst);
free(host_src1);
```

```
    free(host_src2);

    return 0;
}
```

6.2 编译流程

由 Cambricon BANG C 编写的应用程序可以使用 CNCC 进行编译，生成可以在 Cambricon BANG 异构计算平台上运行的可执行程序。CNCC 是 Cambricon BANG C 的语言编译工具，支持主机端和设备端的混合程序的编译，其工作流程如 [图 6.1 CNCC 混合编译流程](#) 所示，CNCC 会对混合了设备端和主机端代码的 *.mlu 做多次编译；面向主机端编译时，设备端的代码会被编译器忽略；面向设备端编译时，主机端的代码会被忽略。

Cambricon@155chb

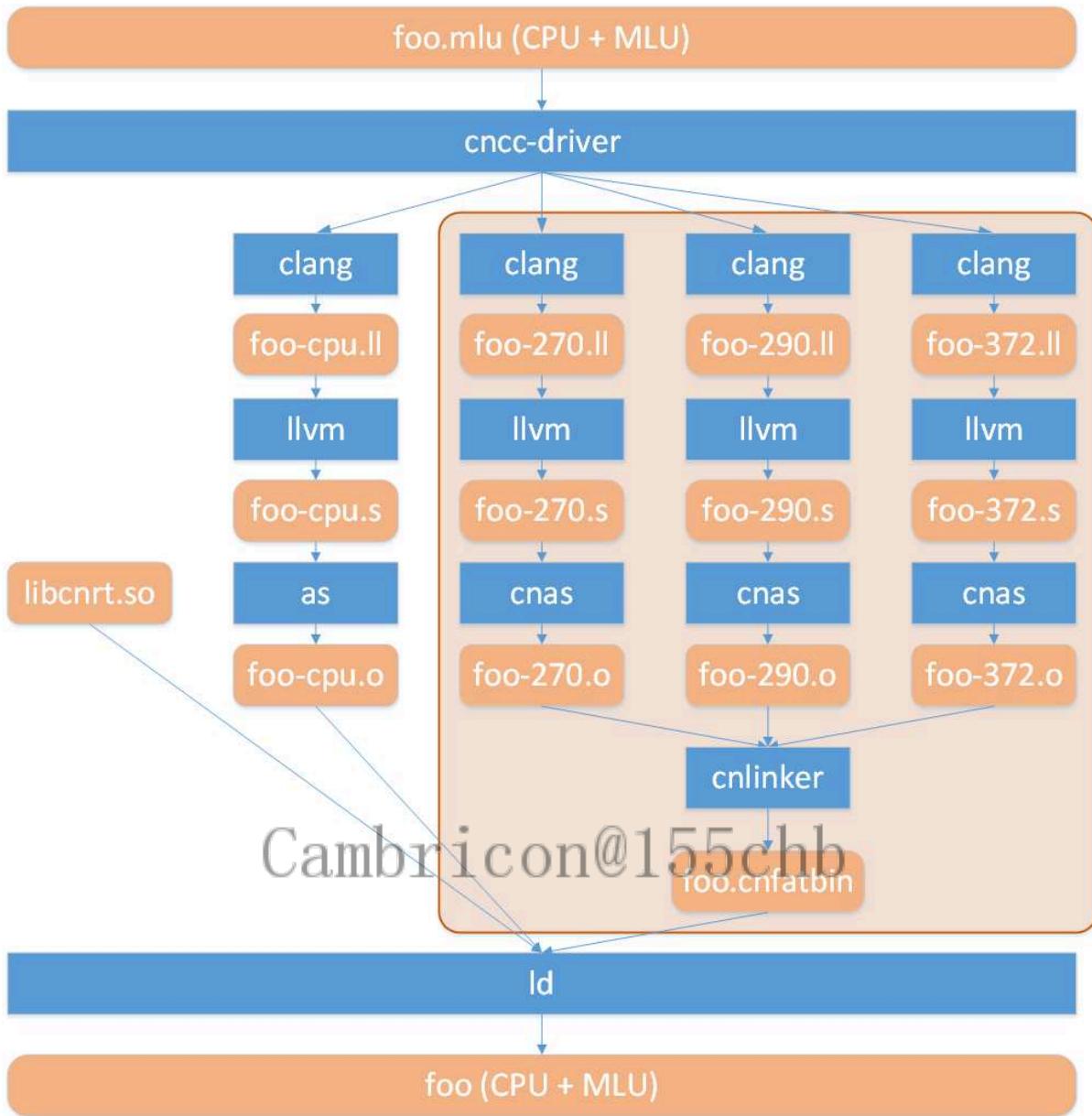


图 6.1: CNCC 混合编译流程

由于在 Cambricon BANG C 编程语言中，设备端程序都有专门的属性标识（详情参见附录C++ 语言扩展），因此 CNCC 可以很容易地区分主机端和设备端代码。无论是面向主机端编译还是面向设备端编译，CNCC 都是先调用 Clang 语言前端将 Cambricon BANG C 程序转换为 LLVM IR。对于主机端来说，CNCC 会调用 LLVM 进行优化并生成对应架构的目标文件。对于设备端来说，CNCC 调用 LLVM 生成面向寒武纪硬件的 MLISA 汇编语言；MLISA 汇编语言经过 CNAS 汇编器的处理得到设备端的目标文件，面向不同设备的目标文件经过链接形成一个 CNFatbin 文件。最后，CNCC 会调用本地的链接器将设备端的 CNFatbin 文件、运行时库以及主机侧的可执行程序链接在一起，形成最终的可执行程序。

设备端的目标文件都是面向特定架构的，也就是说，面向架构 A 编译的目标文件无法在架构 B 的硬件上运行。为此，在编译设备端目标文件时需要通过命令行选项 `--bang-mlu-arch=` 明确指定目标架构；也可以通过 `--bang-arch=` 选项指定属于特定计算能力的所有架构，生成一个可以在一系列架构上运行的

`fatbin` 文件，由运行时环境根据具体的架构选择合适的可执行程序。目前 CNCC 支持的计算能力和具体架构可以参考 [表 4.1 不同计算能力的特性支持情况](#)。

Cambricon BANG 异构并行计算平台可以保证硬件特性的兼容性，例如，面向 `compute_20` 编写的 Cambricon BANG C 程序可以不经修改或者经过少量修改便可以适配 `compute_30`。CNCC 定义了 `--BANG_ARCH__` 宏用于辅助用户编写适应不同架构的程序，这个宏只能用于设备端编程。`--BANG_ARCH__` 宏的数值与目标架构一一对应，例如，当指定的目标架构为 `mtp_372` 时，`--BANG_ARCH__ = 372`。

6.3 CNCC 常见编译选项

在 Linux 系统中，CNCC 以二进制命令的形式供用户使用，其语法格式为：

```
cncc 输入文件名 [-编译选项 1] [-编译选项 2] [.....]
```

常见的编译选项如 [表 6.1 常见的 CNCC 编译选项](#) 所示：

表 6.1: 常见的 CNCC 编译选项

建议编译选项	作用
<code>--help</code>	查看 CNCC 帮助信息。
<code>-o</code> 输出文件名	指定输出目标文件名。
<code>--bang-mlu-arch=...</code>	指定目标架构型号。
<code>--bang-arch=...</code>	指定目标计算能力编号。
<code>-S</code>	输出主机端或者设备端汇编指令。
<code>--bang-device-only</code>	面向设备侧编译程序，通常与 <code>-S</code> 选项配合使用，生成 MLISA 汇编代码。
<code>--bang-host-only</code>	面向主机侧编译程序。
<code>-O3/2/1/0/s</code>	设置编译优化级别（若不加则默认为 <code>-O0</code> ，即不做自动编译优化）。
<code>-g</code>	输出带有调试信息的可执行程序，目前仅能与 <code>-O0</code> 配合使用。

以下是将包含主机侧和设备侧代码的 `main.mlu` 文件编译成可以在 `mtp_270` 架构上运行的可执行程序的命令：

```
cncc main.mlu -o main.out --bang-mlu-arch=mtp_270 -O3
```

6.4 CNBin 与 CNFatbin

CNBin 文件是一种基于 ELF (Executable and Linkable Format, 可执行与可链接格式) 结构的文件。其包含了某一特定架构的设备侧程序代码以及相关的符号、重定位以及调试信息等额外信息。

CNFatbin 文件包含一种或者多种不同架构的 CNBin 文件。CNRT 或者 CNDrv 可以在运行时根据不同的环境加载不同架构的设备侧代码，从而实现一次编译多处运行的功能。

CNCC 支持使用 `--bang-fatbin-only` 选项直接生成 CNFatbin 文件。以下是将包含设备侧代码的 `device_code.mlu` 文件编译成可以在 `mtp_270`、`mtp_372` 以及 `mtp_592` 架构上运行的 CNFatbin 文件的命令：

```
cncc --bang-fatbin-only --bang-mlu-arch=mtp_270 --bang-mlu-arch=mtp_372 --bang-mlu-arch=mtp_
-592 device_code.mlu -o device_code.cnfatbin -03
```

CNFatbin 文件可以由 CNDrv 中 Module 管理系列接口动态加载并执行。具体使用请参考《Cambricon Driver API Developer Guide》。

6.5 CNRT 和 CNDrv 的使用

CNRT (Cambricon Runtime Library, 寒武纪运行时库) 和 CNDrv (Cambricon Driver API, 寒武纪软件栈驱动接口) 包含设备管理、内存管理、任务队列管理、设备端程序执行、通知管理等功能。使用 Cambricon BANG C 编写的程序需要借助于 CNRT 或者 CNDrv 才能运行在 MLU 设备上。

6.5.1 CNRT 和 CNDrv 的区别

CNRT 和 CNDrv 非常相似，在很多情况下是可以互相替换的，并且也支持 CNRT 和 CNDrv API 的混合调用。

CNRT 简化了设备使用的流程，内部实现了 CNDrv 初始化和 Context、Kernel、Module 管理等功能（详情请参考 第 9.4 章 CNDrv API），但是其缺少一些对于设备的管理能力。

CNDrv 提供了更细粒度的资源操作能力，用户可以自由控制 Context 的切换以及 Kernel、Module 在设备上的加/卸载，从而避免 CNRT 在初始化阶段就将所有 Module 加载到设备上。但是对于 Kernel 的启动和参数配置等功能则需要用户调用一些接口完成。

6.5.2 CNRT 的初始化

CNRT 没有显式的初始化函数，采用的是隐式初始化测例，用户在第一次调用 CNRT API 时就会完成对 CNRT 的初始化，具体流程如 [图 6.2 CNRT 的初始化流程](#) 所示。

注解：

- 当用户第一次调用的 CNRT API 不需要激活 Shared Context 时，CNRT 的初始化并不会激活 Shared Context，比如版本管理类 API；
- 当用户第一次调用的 CNRT API 需要激活 Shared Context 时，CNRT 就会激活 Shared Context 并加载 Kernel，比如内存申请类 API。

CNRT 的初始化分为三个阶段：

- 初始化运行环境：这个阶段 CNRT 不会申请设备资源；
- 创建与激活 Shared Context；
- 加载 Kernel Module：这个阶段会申请一部分的设备资源，用于存放 Kernel 运行相关数据和指令。

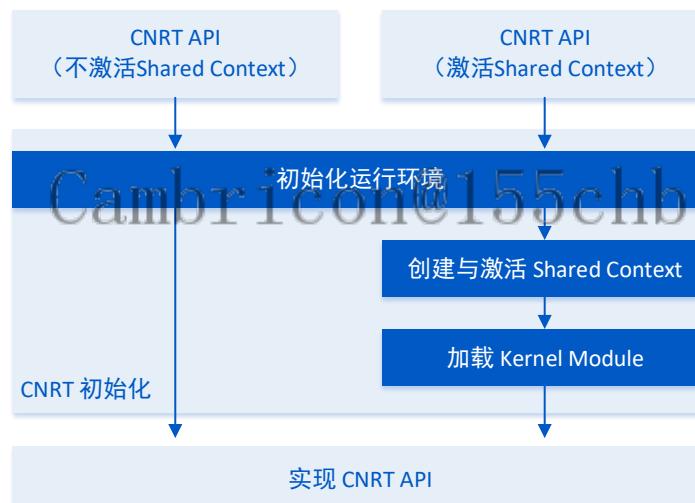


图 6.2: CNRT 的初始化流程

注意：

由于在 CNRT 初始化流程中会创建并激活 Shared Context，同时会加载 Kernel Module 到设备侧。因此，对于任意一个会导致 CNRT 初始化的 API 来说，都有可能存在由于激活 Shared Context 或加载 Kernel Module 失败而返回错误码。

CNRT 初始化时，会为每个 MLU 设备创建一个 Context，该 Context 为此设备的 Shared Context，在所有主机线程之间共享。刚创建的 Shared Context 处于未激活的状态，在第一次调用 CNRT 接口时，该 Shared Context 便会被初始化，并处于激活状态。基于 Shared Context，用户申请的资源可以在 CNRT 与 CNDrv 之间互通操作。

当用户在主机侧线程中调用 `cnrtDeviceReset()` 后，基于当前主机线程的 Shared Context 申请的设备

资源会被销毁。此时，当用户将此设备作为当前设备且在任意一个主机线程中继续调用了 CNRT API，且该 API 需要处于激活状态的 Shared Context 时，该设备的 Shared Context 会重新被激活。

6.5.3 CNDrv 的初始化

与 CNRT 无显式初始化接口不同，在当前系统中初始化 CNDrv 运行环境，需要调用 `cnInit()` 进行设备初始化。该接口可以在进程开始调用，也可以在每个线程开始时调用，但必须在使用 CNDrv 接口前调用。

6.5.4 设备管理接口

由于 CNRT 支持单机多卡运行，在执行设备端程序前需要指定一个 MLU 设备，因此 CNRT 设备管理接口的主要功能就是获取设备数量、指定执行设备、获取设备属性等。

表 6.2: CNRT 常用设备管理接口简介

API 名称	功能简介
<code>cnrtGetDeviceCount()</code>	获取本机 MLU 设备数量
<code>cnrtSetDevice()</code>	指定即将使用的 MLU 设备
<code>cnrtGetDevice()</code>	获取当前使用的设备
<code>cnrtDeviceGetAttribute()</code>	获取设备属性信息
<code>cnrtDeviceReset()</code>	设备重置
<code>cnrtSyncDevice()</code>	设备同步

6.5.5 设备内存管理

MLU 的编程模型假设系统由主机内存和设备内存组成，这两者是独立的内存系统。在设备上运行的 Kernel 程序使用的是设备内存，CNRT 提供了内存申请、释放以及内存数据在主机和设备之间的拷贝功能。设备内存的申请返回的是设备侧连续虚拟地址。这个虚拟地址可以作为指针在不同的 Kernel 函数中传递并由 MLU Core 直接访问。

注意：

- MLU 加速卡平台虚拟地址位宽为 64bit。
- CE(Cambricon Edge) 边缘计算平台虚拟地址位宽为 64bit。

通常情况下，用户需要使用 `cnrtMalloc()` 申请设备内存并使用 `cnrtFree()` 对设备内存进行释放操作。

6.5.5.1 设备内存拷贝

6.5.5.1.1 同步内存拷贝和异步内存拷贝介绍

内存拷贝包括同步拷贝和异步拷贝两种方式。

- 同步拷贝内存数据，支持主机端到设备端、设备端到主机端、设备端到设备端，以及主机端到主机端的数据拷贝，通常使用 `cnrtMemcpy()` 完成。
- 异步拷贝内存数据，适用于拷贝数据量较大的场景。支持主机端到设备端、设备端到主机端、设备端到设备端的数据拷贝，通常使用 `cnrtMemcpyAsync()`。

注意：

- 调用异步拷贝接口时，传入的设备内存和队列必须是在同一个 Context 下申请的，否则接口会报错。
- 调用异步拷贝接口时，用户需要保证在队列同步操作也就是拷贝完成前，设备地址和内存地址不会被释放，否则可能会导致不可预知的后果。

一般为了提高 MLU Core 核心利用率，用户至少需要创建两个线程，一个线程准备数据，另一个线程计算。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，不用等待前一份数据计算完成后，再进行下一份数据的拷贝，因此用户的线程不会再阻塞。

下面这个例子介绍通过拷贝接口完成数据从主机侧到设备侧的拷贝，并交由 MLU Core 进行简单的标量加法操作。

```
#include "bang.h"

__mlu_global__ void mlu_add(float *x, float *y) {
    *y = *x + *y;
}

int main() {
    float *d0, *d1;
    size_t mem_size = sizeof(float);

    cnrtMalloc((void **)&d0, mem_size);
    cnrtMalloc((void **)&d1, mem_size);

    float h0 = 1.0, h1 = 2.0;
    cnrtMemcpy(d0, &h0, mem_size, cnrtMemcpyHostToDev);
    cnrtMemcpy(d1, &h1, mem_size, cnrtMemcpyHostToDev);

    cnrtDim3_t dim;
    dim.x = 1;
```

```
dim.y = 1;
dim.z = 1;

cnrtFunctionType_t type = cnrtFuncTypeBlock;
cnrtQueue_t queue;
cnrtQueueCreate(&queue);

mlu_add<<<dim, type, queue>>>(d0, d1);
cnrtGetLastError();
cnrtSyncDevice();

cnrtFree(d0);
cnrtFree(d1);
cnrtQueueDestroy(queue);

return 0;
}
```

6.5.5.1.2 设备内存多维同步拷贝介绍

对于不连续的内存拷贝，或数据格式化提取等操作，则需要多维内存拷贝。例如规律的跳变搬运，可以使用二维内存拷贝实现。对于三维的数据拷贝或三维数据提取、转换等操作，可以使用三维内存拷贝实现。

二维内存拷贝介绍

支持二维内存拷贝和二维内存转换。同步内存拷贝中，二维内存拷贝用 `cnrtMemcpy2D()` 完成。

二维内存转换

支持一维和二维内存的互相转换，如一维内存转换为二维内存和二维内存转换为一维内存。典型的使用场景如 RGB 三色图的不同色域的提取，可以使用二维内存转换为一维内存，逐一将源内存的 R 色域、G 色域、B 色域的数据提取到对应的一维内存中。同理可以将处理后的 R 色域、G 色域、B 色域的数据，通过一维内存转换为二维内存的方式，合成 RGB 三色图。

如图 6.3 二维内存转换为一维内存 所示，提取 8 列（width）共 4 行（height）数据，顺序的存储到目的内存。



图 6.3: 二维内存转换为一维内存

```

int memcpy_2d_to_1d_example(void)
{
    uint8_t *cnAddrSrc;
    uint8_t *cnAddrDst;

    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrSrc, size));
    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrDst, size));

    /* 预处理源数据和目的数据 */
    .....

    /* 初始化二维拷贝参数 */
    u64 dpitch = 0x8; /* 和 width 相等, 表示从目的地址顺序写入数据 */
    u64 spitch = 0x10; /* 搬运 width 字节后的下一次读取数据的位置 */
    u64 width = 0x8; /* 二维搬运的宽度 */
    u64 height = 0x4; /* 二维搬运的高度 */

    ASSERT_EQ(cnrtSuccess,
              cnrtMemcpy2D(cnAddrDst, dpitch, cnAddrSrc, spitch,
                          width, height, cnrtMemcpyDevToDev));

    /* 释放资源 */
    .....

}

```

一维内存转换为二维内存，是将源内存按照一维的方式连续读取，再按照指定的 width 和 pitch 参数，二维的存放到目的内存里。其中 width 表示一次实际搬运的长度，相当于一维的长度。pitch 表示对齐的大小，相当于二维的跳变，搬运 width 字节后下一次操作数据的位置。

```
int memcpy_1d_to_2d_example(void)
{
    uint8_t *cnAddrSrc;
    uint8_t *cnAddrDst;

    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrSrc, size));
    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrDst, size));

    /* 预处理源数据和目的数据 */
    .....

    /* 初始化二维拷贝参数 */
    uint64_t dpitch = 0x10; /* 搬运 width 字节后的下一次写入数据的位置 */
    uint64_t spitch = 0x8; /* 和 width 相等，表示从源地址顺序读取数据 */
    uint64_t width = 0x8; /* 二维搬运的宽度 */
    uint64_t height = 0x4; /* 二维搬运的高度 */

    ASSERT_EQ(cnrtSuccess,
              cnrtMemcpy2D(cnAddrDst, dpitch, cnAddrSrc, spitch,
                          width, height, cnrtMemcpyDevToDev));

    /* 释放资源 */
    .....

}
```

Cambricon@155chb

二维内存拷贝

二维内存到二维内存的搬运，典型的应用场景如图层的叠加，可以将处理好的图层叠加到目的图片上，实现图像的快速处理。可以将预处理好的 R 色域图层，按照相同的规则（spitch=dpitch）读取和写入到对应的目的图层上。

如图 6.4 二维内存拷贝 所示：

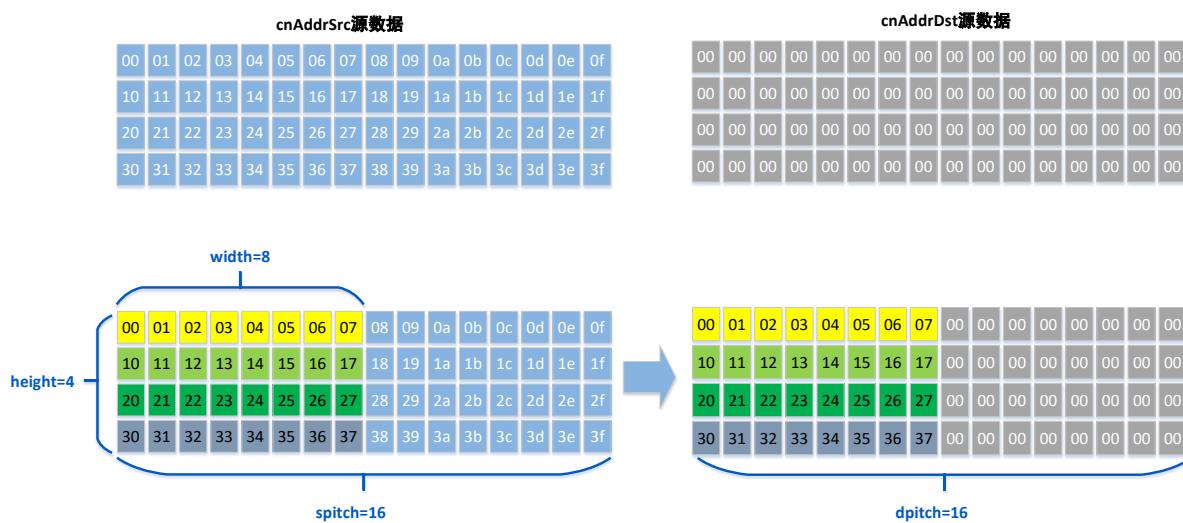


图 6.4: 二维内存拷贝

```

int memcpy_2d_test_example1(void)
{
    uint8_t *cnAddrSrc;
    uint8_t *cnAddrDst;

    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrSrc, size));
    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrDst, size));

    /* 预处理源数据和目的数据 */
    .....

    /* 初始化二维拷贝参数 */
    u64 dpitch = 0x10; /* 搬运 width 字节后, 下一次写入数据的位置 */
    u64 spitch = 0x10; /* 搬运 width 字节后, 下一次读取数据的位置 */
    u64 width = 0x8;   /* 二维搬运的宽度 */
    u64 height = 0x4;  /* 二维搬运的高度 */

    ASSERT_EQ(cnrtSuccess,
              cnrtMemcpy2D(cnAddrDst, dpitch, cnAddrSrc, spitch,
                          width, height, cnrtMemcpyDevToDev));

    /* 释放资源 */
    .....

}

```

三维内存拷贝介绍

支持三维内存拷贝和三维内存转换。同步内存拷贝中，三维内存拷贝用 `cnrtMemcpy3D()` 完成。

三维内存转换

支持一维和三维内存的互相转换。例如将三维立体图中的一个子集提取到一维内存，或将处理后的一维数据，恢复到三维立体图中。

三维内存转换为一维内存，首先对源内存读取 `width` 字节数据，顺序写入目的内存。然后跳 `srcPtr.pitch` 到第二个位置继续搬运下一个 `width` 字节数据，顺序写入目的内存。循环 `height` 次，完成第一个二维内存的搬运。然后跳 `srcPtr.pitch * srcPtr.ysize` 开始搬运下一个二维内存，循环 `depth` 次二维搬运，从而完成三维内存的拷贝。

```
int memcpy_3d_to_1d_example(void)
{
    uint8_t *cnAddrSrc;
    uint8_t *cnAddrDst;
    cnrtMemcpy3dParam_t p = {0};
    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrSrc, size));
    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrDst, size));
    /* 预处理源数据和目的数据 */
    .....

    /* 初始化三维拷贝参数 */
    p.dstPtr.pitch = 0x8; /* 和 width 相等，表示从目的地址顺序写入数据 */
    p.dstPtr.xsize = 0x10; /* 目的内存的 x 方向长度 */
    p.dstPtr.ysize = 0x4; /* 目的内存的 y 方向长度，ysize*pitch 是三维 stride 的长度 */

    p.extent.depth = 0x2; /* 三维搬运的深度 */
    p.extent.height = 0x2; /* 三维搬运的高度 */
    p.extent.width = 0x8; /* 三维搬运的宽度 */

    p.srcPtr.pitch = 0x10; /* 搬运 width 字节后，下一次读取数据的位置 */
    p.srcPtr.xsize = 0x10; /* 源内存的 x 方向长度 */
    p.srcPtr.ysize = 0x4; /* 源内存的 y 方向长度，ysize*pitch 是三维 stride 的长度 */

    p.srcPtr.ptr = (void *)cnAddrSrc;
    p.dstPtr.ptr = (void *)cnAddrDst;

    ASSERT_EQ(cnrtSuccess, cnrtMemcpy3D(&p));
```

```
/* 释放资源 */
.....
}
```

三维内存拷贝

使用三维拷贝，可以在 $8 \times 8 \times 8$ 的立方体中，提取出一个 $4 \times 4 \times 4$ 的子立方体。首先对源内存读取4字节数据，写入对应的目的内存。然后跳8字节到第二个位置继续搬运下一个4字节数据。循环4次，完成立方体第一个面的搬运。然后跳 8×8 字节开始搬运立方体的下一个面，循环4次完成子立方体的提取。

如图 6.5 三维内存拷贝 所示：

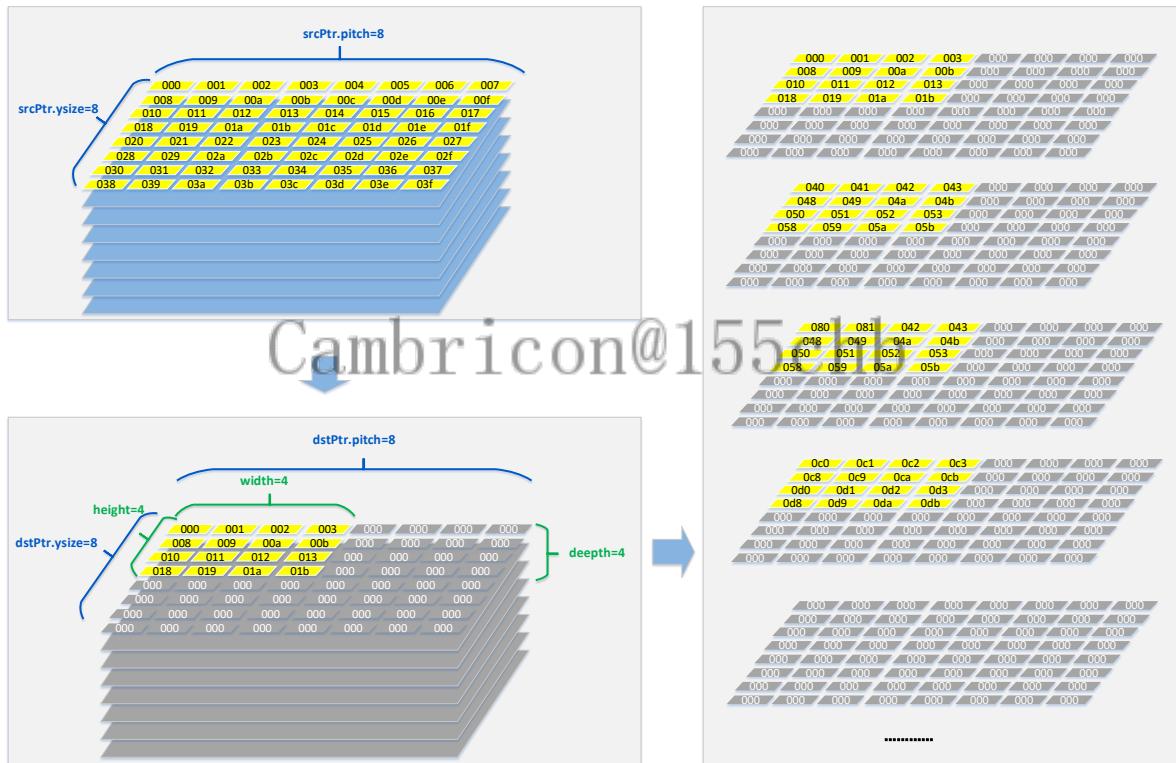


图 6.5: 三维内存拷贝

```
int memcpy_3d_cube_example(void)
{
    uint8_t *cnAddrSrc;
    uint8_t *cnAddrDst;
    cnrtMemcpy3dParam_t p = {0};
    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrSrc, size));
    ASSERT_EQ(cnrtSuccess, cnrtMalloc((void **)&cnAddrDst, size));
```

```
/* 预处理源数据和目的数据 */
.....
/* 初始化三维拷贝参数 */
p.dstPtr.pitch = 0x8; /* 目的立方体的宽度为 8 */
p.dstPtr.xsize = 0x8; /* 目的立方体的 x 方向长度 */
p.dstPtr.ysize = 0x8; /* 目的立方体的 y 方向长度, ysize*pitch 是三维 stride 的长度 */

p.extent.depth = 0x4; /* 子立方体的深度为 4 */
p.extent.height = 0x4; /* 子立方体的高度为 4 */
p.extent.width = 0x4; /* 子立方体的宽度为 4 */

p.srcPtr.pitch = 0x8; /* 源立方体的宽度为 8 */
p.srcPtr.xsize = 0x8; /* 源立方体的 x 方向长度 */
p.srcPtr.ysize = 0x4; /* 源立方体的 y 方向长度, ysize*pitch 是三维 stride 的长度 */

p.srcPtr.ptr = (void *)cnAddrSrc;
p.dstPtr.ptr = (void *)cnAddrDst;

ASSERT_EQ(cnrtSuccess, cnrtMemcpy3D(&p));
/* 释放资源 */
.....
}
```

Cambricon@155chb

6.5.5.2 云侧 MLU 平台设备内存操作介绍

6.5.5.2.1 MLU 加速卡平台设备 NUMA 内存管理

NUMA (Non-Uniform Memory Access, 非一致性内存访问) 架构可以理解为每个处理器都有本地内存模块，能够直接进行访问。如果数据驻留在远端内存，访问速度就会慢一些，总体上来说，距离处理器越远的内存访问成本越高。在 NUMA 模式中，每个节点有各自的本地内存模块，因此内存访问能够避免与共享内存总线相关的吞吐限制和争用问题；同时，处理器之间通过共享总线（或其他互连形式）访问其他处理器的内存模块。

对于加速卡形态产品，由于芯片架构不同，部分产品设备内存是基于 NUMA 架构进行管理的。对于通过 NUMA 架构进行管理的，可通过 CNDrv 提供的 `cnDeviceGetAttribute()` 接口获取设备 NUMA 信息，与此同时，CNDrv 提供了一组带 NUMA node 的申请设备内存的接口。

下面这个例子介绍通过 CNDrv 提供的 `cnMallocNode()` 接口指定 NUMA node 申请设备内存并下发 Kernel 完成计算的实例。

```
#include "bang.h"

__mlu_global__ void mlu_add(int *x, int *y, uint32_t size) {
    for (uint32_t i = 0; i < size; ++i) {
        y[i] = x[i] + y[i];
    }
}

int main() {
    /* CNRT API must be called first when CNDrv and CNRT are mixed. */
    unsigned int dev_num = 0;
    cnrtGetDeviceCount(&dev_num);
    if (dev_num < 1) {
        printf("There is no device in current system.\n");
        return -1;
    }

    int node_count = 0;
    CNdev device;
    cnDeviceGet(&device, 0);
    /* get NUMA node attribute */
    cnDeviceGetAttribute(&node_count, CN_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_NODE_COUNT, device);

    int *d0, *d1;
    size_t mem_size = 1024 * sizeof(int);

    /* CNresult cnMallocNode(CNaddr *pmluAddr, cn_uint64_t bytes, int node) */
    /* The NUMA node is in the range [0, node_count - 1]. */
    /* Take '2' as 'node' in this sample and then alloc device memory by specified NUMA node */
    cnMallocNode((void **)&d0, mem_size, 2);
    cnMallocNode((void **)&d1, mem_size, 2);

    int h0[1024], h1[1024];
    for (int i = 0; i < 1024; i++) {
        h0[i] = 1;
        h1[i] = 2;
    }

    cnrtMemcpy(d0, h0, mem_size, cnrtMemcpyHostToDev);
    cnrtMemcpy(d1, h1, mem_size, cnrtMemcpyHostToDev);
```

```
cnrtDim3_t dim;  
dim.x = 1;  
dim.y = 1;  
dim.z = 1;  
  
cnrtFunctionType_t type = cnrtFuncTypeBlock;  
cnrtQueue_t queue;  
cnrtQueueCreate(&queue);  
  
mlu_add<<<dim, type, queue>>>(d0, d1, mem_size / sizeof(int));  
cnrtGetLastError();  
cnrtSyncDevice();  
  
cnrtFree(d0);  
cnrtFree(d1);  
cnrtQueueDestroy(queue);  
  
return 0;  
}
```

Cambricon@155chb

6.5.5.3 CE 边缘计算平台设备内存操作介绍

6.5.5.3.1 CE 边缘计算平台内存模型基础与缓存一致性

对于 CE 边缘计算产品，物理内存是一块连续地址空间，但是在系统资源分配上将其分为两部分：

- 操作系统可见内存，即 OS 内存（Operating System Memory）；
- 操作系统不可见内存，即设备内存（Device Memory），由 CNRT 进行管理。

内存模型的基本概念如下图所示：

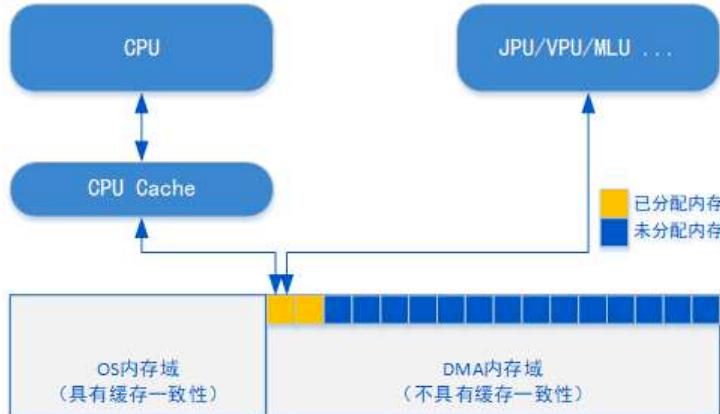


图 6.6: CE 边缘计算内存架构图

注解：

- JPU (JPEG Processing Unit, 图片处理单元)；
- VPU (Video Processing Unit, 视频处理单元)。

从图中可以看出，CPU 对内存的访问是经过缓存的，而设备（JPU/VPU/MLU Core 等）则没有缓存作为缓冲。这种访问方式与 DMA 内存的访问方式较为相似，因此本文选择了 DMA 内存模型作为基础模型来介绍内存框架。

6.5.5.3.2 内存模型与缓存一致性

CPU 进行内存写入时有两种方式：

- 透写 (Write Through)：数据同时更新到缓存和内存中，该方式更加简单、可靠，主要用于无需频繁写入缓存的场景。
- 回写 (Write Back)：数据仅在缓存中更新，只有当缓存行 (Cache Line) 准备好被替换后，数据才会更新到内存中。

CPU 读取内存的情况与写入类似。

设备通过 SoC 的内部总线访问物理内存时不需要经过 CPU，也就没有 CPU 的缓存缓冲功能。

考虑如下情况：

Cambricon@155chb

1. MLU Core 将计算结果数据写入到物理内存中，但是此时缓存中对应该物理内存的缓存行保存了之前的数据，并且状态为有效，因此缓存的硬件无法感知当前物理内存中的数据已经被 MLU Core 改写（红叉路径）；
2. CPU 读取 MLU Core 写入的物理内存，但是由于缓存的存在，CPU 获取到的是缓存存储的旧数据；
3. CPU 解析 MLU Core 输出，由于数据不正确而解析失败。

该访问流程如下图所示：

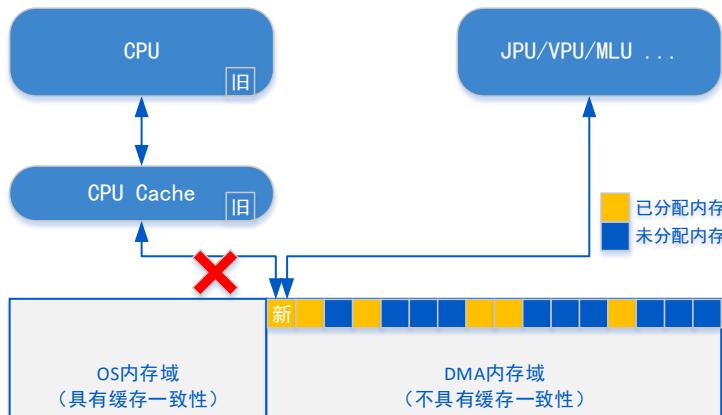


图 6.7: 缓存一致性示意图

6.5.5.3.3 缓存一致性维护

为了保证 CPU 和设备之间内存访问的数据一致性，需要遵循一定的访问规则，如下图所示。

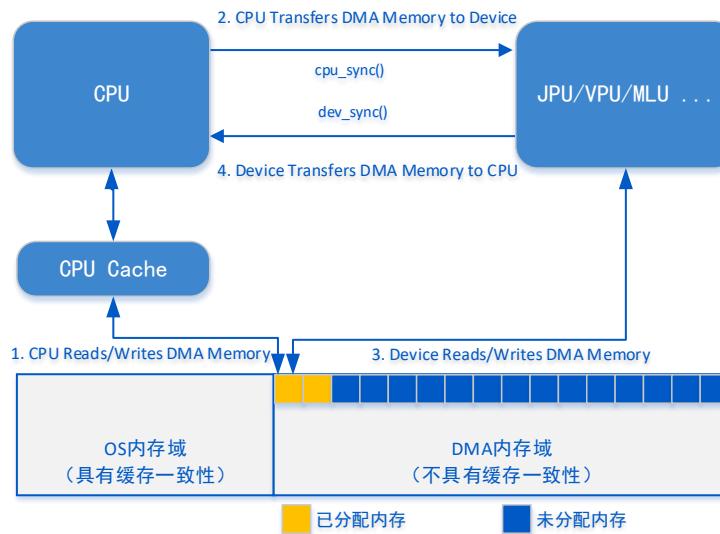


图 6.8: 缓存一致性维护示意图

提示：

为便于描述，`cpu_sync()` 和 `dev_sync()` 在本节只作为抽象接口来描述缓存的同步操作，并非实际接口名称。CNRT 提供了 CPU 访问后与设备进行内存数据同步的接口，以及在设备访问后与 CPU 同步内存数据的接口，具体接口使用描述请参见 第 6.5.5.3.6 章节 CE 平台缓存操作。

如 图 6.8 缓存一致性维护示意图 所示，访问过程时的数据一致性保证分为 CPU 访问和设备访问两个部分：

- CPU 对 DMA 内存进行读/写：在操作 API 启动设备前，调用 `cpu_sync()` 同步接口保证 CPU 的缓存内容与 DMA 内存一致；
- 设备对 DMA 内存进行读/写：在 API 返回后，调用 `dev_sync()` 同步接口保证设备内存操作已经与缓存一致。

同步接口 `cpu_sync()` 和 `dev_sync()` 操作的最小单位是一个缓存行（Cache Line，CE 平台上是 64 字节）。当 CPU 和设备操作同一块缓存行对齐的物理内存时，需要维护缓存一致性。如果 CPU 或者设备访问缓存行的前半部分，设备或者 CPU 访问缓存行的后半部分，两个地址不重叠，也需要维护缓存一致性。有些变量是原子类型以及包含原子变量的结构体，例如互斥锁，信号量，读写锁等，这些变量的内存是缓存属性的，它们访问的都是 CPU Cache，不能使用具有缓存属性的 DMA 内存域地址。其他类型的变量也不能使用具有缓存属性的 DMA 内存域地址。

无法保证缓存一致性的错误做法：

- 错误做法一：将具有缓存属性的 DMA 内存域地址 `cache_dma_addr` 强转成原子变量的地址。

```
atomic<int> *p_atomic_val = (atomic<int> *)cache_dma_addr;
```

- 错误做法二：将具有缓存属性的 DMA 内存域地址 cache_dma_addr 强转成整型变量的地址。

```
unsigned int *p_u32_val = (unsigned int *)cache_dma_addr;
```

- 错误做法三：将具有缓存属性的 DMA 内存域地址 cache_dma_addr 强转成包含原子变量的互斥锁地址。

```
pthread_mutex_t *p_mutex_lock = (pthread_mutex_t *)cache_dma_addr;
```

6.5.5.3.4 CE 平台设备内存申请与释放

本小节介绍 CE 平台设备内存申请与释放，包括普通设备内存申请与扩展设备内存申请。

设备内存申请与释放的通用方法

申请设备内存

该接口为端侧、云侧统一形式的设备内存申请接口，用于申请指定 bytes 大小的 MLU 设备内存。申请成功返回 cnrtSuccess，否则返回相应的错误码。

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

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 cnrtSuccess，否则返回相应的错误码。

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

注意：

- 建议解映射后再调用 cnrtFree，释放已经映射过的设备内存；虽然 cnrtFree 不会限制用户释放未进行解映射的内存，但在释放前进行解映射是很好的编程习惯。
- cnrtFree 不能释放从系统申请到的内存，也不能释放映射后的主机侧内存。

6.5.5.3.5 CE 平台内存映射与解映射

建立指定范围的设备侧内存到无缓存属性的用户态主机侧内存的映射

将输入参数 dev_ptr 指向的设备侧的内存空间，按照输入 size 的大小与 pHostPtr 对应的无缓存属性的内存进行映射。

```
cnrtRet_t cnrtMmap(void *ptr, void **pHostPtr, size_t size);
```

注意：

请避免使用 cnrtMmap 对某一块固定的设备内存进行反复映射。由于 cnrtMmap 接口多次调用的耗时与调用次数成正比，因此，为降低耗时，请减少对同一块设备内存的映射次数。

```
/* 错误的使用方式：对同一块设备内存反复映射 */
void *dev_ptr;
size_t s = 0x100000;
cnrtMalloc(&dev_ptr, s);
void *host_ptr[MAP_REPEAT];
for (i = 0; i < MAP_REPEAT; i++) {
    cnrtMmap(dev_ptr, &host_ptr[i], s);
    access_host_ptr(host_ptr[i]);
    .....
}

/* 正确的使用方式：每次映射一部分，访问读取时只使用映射过的部分 */
void *dev_ptr;
size_t s = 0x100000;
int div = 16;
cnrtMalloc(&dev_ptr, s);
void *host_ptr[MAP_DEVIDE * div];
for (i = 0; i < MAP_DEVIDE * div; i++) {
    cnrtMmap(dev_ptr + i * (s/div), &host_ptr[i], s/div);
    access_host_ptr(host_ptr[i]);
    .....
}
```

建立指定范围的设备侧内存到具有缓存属性的用户态的主机侧地址的映射

将输入参数 `dev_ptr` 指向的 MLU 端的内存空间，按照输入 `size` 的大小与 `pHostPtr` 对应的具有缓存属性的内存进行映射。

```
cnrtRet_t cnrtMmapCached(void *ptr, void **pHostPtr, size_t size);
```

注意：

- 具有缓存属性的内存在使用的时候，要注意缓存的一致性。当读取的数据出现概率性的不一致时，可能是缓存一致性维护的问题，此时建议换为无缓存属性的内存进行比对测试。
- 不建议对同一块地址多次映射，多次映射会增加驱动合法性检查的时间，此外系统有映射次数限制，次数太多会失败；每次映射会消耗系统内存，如果多次映射超级大的内存，会导致系统内存溢出。
- 如果需要多次建立映射，建议减少映射次数，尽量避免对同一段设备侧内存做多次映射。
- `cnrtMmap` 和 `cnrtMmapCached` 是独立接口，它们之间没有联系，建议先使用 `cnrtMmap` 接口把功能调对，之后用 `cnrtMmapCached` 接口调性能。因为相比具有缓存属性的内存，无缓存属性的内存虽然不需要维护缓存一致性，但是会有较差的性能。
- 当用户调用 `cnrtMmap` 和 `cnrtMmapCached` 接口传入的 `size` 超过申请时的 `size` 时，接口会返回错误。

Cambricon@155chb

指定大小的内存解映射

输入 `hostPtr` 参数指向需要解映射的内存虚拟地址，按照输入的 `size` 取消 `cnrtMmap` 或 `cnrtMmapCached` 建立的内存虚拟地址映射。

```
cnrtRet_t cnrtMunmap(void *hostPtr, size_t size);
```

注意：

- `cnrtMmap` 必须使用 `cnrtMunmap` 进行对应的解映射；
- 无缓存属性的内存不需要关注缓存的一致性。但相比具有缓存属性的内存，无缓存属性的内存使用时会有较差的性能。

CE 平台升级指导

1. cnrtMap 建议由 cnrtMmapCached 替换，旧接口 cnrtMap 后的地址默认为缓存的，新接口 cnrtMmapCached 可以实现这个功能。替换风险：相比旧的接口会有一定程度性能下降。
2. cnrtUnmap 建议由 cnrtMunmap 替换。该新接口需要与 cnrtMmap 或 cnrtMmapCached 配合使用。

6.5.5.3.6 CE 平台缓存操作

CNRT 提供了在 CE 边缘平台对内存进行缓存操作的接口。

缓存的刷新或使无效

缓存操作接口

根据输入的 opr 对 hostPtr 指向的缓存地址按照输入的 size 大小进行刷新或使其无效的操作。

```
cnrtRet_t cnrtMCacheOperation(void *ptr, void *hostPtr, size_t size, cnrtCacheOps_t opr);
```

CE 平台的升级指导

旧接口 cnrtCacheOperation 以及 cnrtCacheOperationRange 可由 cnrtMcacheOperation 替代。cnrtMcacheOperation 除了支持缓存的刷新 (flush) 操作之外，还支持使缓存失效 (invalidate) 的操作。此外，用户调用 cnrtMmap 或 cnrtMmapCached 获取的地址只能通过 cnrtMCacheOperation 进行相应操作。

注意：

- 用户要注意维护缓存一致性。当缓存由 CPU 写入后作为 MLU Core 的输入，此时如果用户忘记调用 cnrtMCacheOperation 对这块内存进行刷新，就会导致 MLU Core 没有读取到正确数值。比较典型的现象是，MLU Core 计算时而正确时而错误，但是出错的时候，CPU 读回的输入数据却一直是正确的。
- 由于没有维护缓存一致性，还会导致读取不到正确的数据。例如当 CPU 在其他设备写入 DDR 数据前读取了地址内数据，导致其他 CPU 线程获取不到正确的计算结果。

提示：

- 调用 `cnrtMCacheOperation` 接口传入的 `size` 需要小于或等于实际的内存大小，否则接口会直接返回合法性检查错误，此时不会进行缓存相关操作。
- 调用 `cnrtMcacheOperation` 接口传入的 `ptr` 和 `hostPtr` 要一一匹配，否则接口会直接返回合法性检查错误。
- 用户通过旧接口 `cnrtMap` 获取的内存不能使用 `cnrtMCacheOperation` 做缓存操作。`cnrtMCacheOperation` 需要传入的地址是通过 `cnrtMmapCached` 获取的具有缓存属性的内存。

6.5.5.3.7 CE 平台缓存内存使用建议

相比具有缓存属性的内存，无缓存属性的内存读写性能会差一些。CPU 对内存写操作做了优化，并且由于缓存属性的内存多了 `cnrtMcacheOperation` 刷新内存操作，导致缓存属性内存的写操作不一定比无缓存属性内存的写操作性能高。CNToolkit Sample 提供了 `cacheWritePerformance` 例子，通过给定不同的拷贝大小，可以找到缓存属性和非缓存属性带宽的大概交点，根据交点处的拷贝大小指导用户选择内存属性类型。例如多次运行测例，如果发现当数据量小于 32KB 时，即使性能有波动，但总体上，缓存内存的写性能低于无缓存内存的写性能；当数据量大于 32KB 时，缓存内存的写性能永远高于无缓存内存的写性能。那么当 CPU 只有写操作，没有读操作，并且数据量小于 32KB 时，可以选择用无缓存属性的内存，否则使用缓存属性的内存。

6.5.5.3.8 CE 平台内存信息查询

内存属性查询

获取内存地址的属性

```
cnrtRet_t cnrtPointerGetAttributes(cnrtPointerAttributes_t *attr, const void *ptr);
```

`cnrtPointerGetAttributes` 接口可以一次性获取输入地址的所有信息，便于用户查询以及进行相关判断。`deviceBaseAddr` 属性可以获取设备地址的基地址，`size` 可以返回申请内存的大小，这两个属性可用于替代旧接口 `cnrtFindDevAddrByMappedAddr` 与 `cnrtGetMemorySize` 的功能；`devicePointer` 属性可以获取映射前的设备地址，可以用于替代旧接口 `cnrtFindDevAddrWithOffsetByMappedAddr`。

使用与升级实例

1. cnrtFindDevAddrByMappedAddr 当前仅仅作为兼容性接口，用户可调用 cnrtPointerGetAttributes 并读取 attr 中的 deviceBaseAddr 将其替代。

```
void _REPLACE_cnrtFindDevAddrByMappedAddr(void *host_ptr, void **dev_ptr) {
    cnrtPointerAttributes_t attr;
    cnrtPointerGetAttributes(&attr, host_ptr);
    *dev_ptr = attr.deviceBaseAddr;
}
```

2. cnrtFindDevAddrWithOffsetByMappedAddr 当前仅仅作为兼容性接口，用户可调用 cnrtPointerGetAttributes 并读取 attr 中的 devicePointer 将其替代。

```
/* 获取一块映射后的主机内存 */
void _REPLACE_cnrtFindDevAddrWithOffsetByMappedAddr(void *host_ptr, void **dev_ptr) {
    cnrtPointerAttributes_t attr;
    cnrtPointerGetAttributes(&attr, host_ptr);
    *dev_ptr = attr.devicePointer;
}
```

3. cnrtGetMemorySize 的功能可调用 cnrtPointerGetAttributes 接口后通过查询属性中的 deviceBaseAddr 和 size 来替代。

```
/* 根据输入的设备地址查询申请的起始地址和大小 */
void _REPLACE_cnrtGetMemorySize(void **devBasePtr, void *devPtr, size_t *size) {
    cnrtPointerAttributes_t attr;
    cnrtPointerGetAttributes(&attr, host_ptr);
    *devBasePtr = attr.deviceBaseAddr;
    *size = attr.deviceAddrSize;
}
```

4. 判断输入的设备地址是否合法。

```
/* 若合法返回 0， 不合法或其他错误返回-1， 不是设备地址返回 1 */
int judge_device_addr_sanity(void *ptr) {
    cnrtPointerAttributes_t attr;
    if (cnrtPointerGetAttributes(&attr, ptr)) {
        printf("unknown error.\n");
        return -1;
    }
    if (cnrtMemTypeUnregistered == attr.type) {
        return -1;
    } else if (cnrtMemTypeHost == attr.type) {
```

```

    return 1;
} else if (cnrtMemTypeDevice == attr.type) {
    return 0;
} else {
    printf("unknown error.\n");
    return -1;
}
}
}

```

5. 判断某个地址是否是具有缓存属性的地址。

```

/* 若是返回 0, 不是则返回 1, 其他错误返回-1*/
int judge_addr_is_mapped(void *ptr) {
    cnrtPointerAttributes_t attr;
    if (cnrtPointerGetAttributes(&attr, ptr)) {
        printf("unknown error.\n");
        return -1;
    }
    if (cnrtUvaCached == attr.cacheMode) {
        return 0;
    } else if (cnrtUvaUnCached == attr.cacheMode) {
        return 1;
    } else {
        printf("unknown error.\n");
        return -1;
    }
}

```

Cambricon@155chb

6.5.5.3.9 CE 平台升级指导汇总

表 6.3: 新旧接口对比汇总

旧版本 API 名称	新版本替代的 API 名称
cnrtCacheOperation(void *host_ptr, cnrtCacheOps_t opr)	cnrtMcacheOperation(void *ptr, void *hostPtr, size_t size, cnrtCacheOps_t ops)
cnrtCacheOperationRange(void *host_ptr, size_t size, cnrtCacheOps_t opr)	cnrtMcacheOperation(void *ptr, void *hostPtr, size_t size, cnrtCacheOps_t ops)

下页继续

表 6.3 – 续上页

旧版本 API 名称	新版本替代的 API 名称
cnrtFindDevAddrByMappedAddr (void *mapped_host_ptr, void **dev_ptr)	cnrtPointerGetAttributes (cnrtPointerAttributes_t *attr, const void *ptr)
cnrtFindDevAddrWithOffsetByMappedAddr (void *mapped_host_ptr, void **dev_ptr)	cnrtPointerGetAttributes (cnrtPointerAttributes_t *attr, const void *ptr)
cnrtGetMemInfo(size_t *free, size_t *total, cnrtChannelType_t channel)	cnrtMemGetInfo (size_t *free, size_t *total)
cnrtGetMemorySize(void **devBasePtr, void *devPtr, size_t *bytes)	cnrtPointerGetAttributes (cnrtPointerAttributes_t *attr, const void *ptr)
cnrtMap (void **host_ptr, void *dev_ptr)	cnrtMmapCached (void *ptr, void **pHostPtr, size_t size)
cnrtMapRange(void **host_ptr, void *dev_ptr, size_t size)	cnrtMmapCached (void *ptr, void **pHostPtr, size_t size)
cnrtUnmap(void *host_ptr)	cnrtMunmap(void *hostPtr, size_t size)

6.5.6 设备 L2 Cache 管理

当前版本 CNDrv 不开放 L2 Cache 管理功能，只允许内部基础软件平台使用此资源。

6.5.7 主机侧页锁定机制内存管理

CNRT 提供了使用页锁定机制（Page-Locked）的主机内存管理接口：

- `cnrtHostMalloc()` 和 `cnrtFreeHost()`：申请和释放页面锁定主机内存；
- `cnrtAcquireMemHandle()`，`cnrtMapMemHandle()`，`cnrtUnMapMemHandle()`：页锁定内存的进程间共享需要使用设备内存的进程间共享接口来完成。

注解：

页锁定机制内存管理不同于通过 `malloc()` 接口申请的普通可分页主机内存。

使用页锁定主机内存有以下几点好处：

- 任务并行：在页锁定主机内存和设备内存之间进行数据拷贝可以和计算任务执行同时进行；
- 带宽优化：由于页锁定主机内存和设备内存之间拷贝，不需要对主机内存进行锁页操作，其带宽要优于普通可分页主机内存和设备内存之间拷贝的带宽。

注意：

- 页锁定内存是一种稀缺的内存资源。相较于普通可分页内存，页锁定内存更容易申请失败。
- 由于通过占用大量的主机物理内存来实现页锁定，所以占用较多的页锁定内存会降低操作系统的整体性能。
- 页锁定内存分配时默认是可缓存的，目前没有提供接口修改缓存属性。

```
#define LOOP_COUNTS (1000)

static double __get_bandwidth(size_t size, float us)
{
    return (((size * LOOP_COUNTS) / us) * 1000000) / (1UL << 20);
}

int main(void) {
    size_t maxSz = 1UL << 28;
    cnrtNotifier_t notifier_s, notifier_e;
    cnrtQueue_t queue;
    float us;

    char *hostMem = NULL;
    char *hostPinnedMem = NULL;
    void *devMem = NULL;

    cnrtNotifierCreate(&notifier_s);
    cnrtNotifierCreate(&notifier_e);

    cnrtQueueCreate(&queue);

    hostMem = (char *)malloc(maxSz);
    cnrtHostMalloc((void **)&hostPinnedMem, maxSz);
    cnrtMalloc(&devMem, maxSz);

    memset(hostPinnedMem, 0x2c, maxSz);
    memset(hostMem, 0x3c, maxSz);

    cnrtPlaceNotifier(notifier_s, queue);
    for (int i = 0; i < LOOP_COUNTS; i++) {
        cnrtMemcpyAsync(devMem, hostMem, maxSz, queue, cnrtMemcpyHostToDev);
    }
    cnrtPlaceNotifier(notifier_e, queue);
```

```
cnrtQueueSync(queue);
cnrtNotifierDuration(notifier_s, notifier_e, &us);
printf("Pageable Host Memory bandwidth: %lfMB/s\n", __get_bandwidth(maxSz, us));

cnrtPlaceNotifier(notifier_s, queue);
for (int i = 0; i < LOOP_COUNTS; i++) {
    cnrtMemcpyAsync(devMem, hostPinnedMem, maxSz, queue, cnrtMemcpyHostToDev);
}
cnrtPlaceNotifier(notifier_e, queue);

cnrtQueueSync(queue);
cnrtNotifierDuration(notifier_s, notifier_e, &us);
printf("Page-locked Host Memory bandwidth: %lfMB/s\n", __get_bandwidth(maxSz, us));
return 0;
}
```

6.5.8 异步并行执行

寒武纪底层软件支持以下操作作为相互独立的任务并行执行:

- 主机上的计算任务;
- 设备上的计算任务;
- 主机到设备的数据拷贝;
- 设备到主机的数据拷贝;
- 设备之间的数据拷贝。

6.5.8.1 主机和设备之间的并发执行

CNRT 提供的异步接口支持异步执行设备任务。调用这些异步接口，设备上执行的任务会加入到任务队列，待设备资源满足需求再执行。主机线程可以在执行这些异步任务同时处理其他任务。下述操作相对于主机是可以异步执行：

- Kernel 任务的启动和执行。
- 单一设备内部内存数据拷贝。
- 设备和主机间通过调用 Async 后缀的接口进行内存数据拷贝。
- 通过调用 Async 后缀的接口进行 Memory set。

6.5.8.2 Kernel 并发执行

如果 Kernel 任务所需要的计算资源满足要求，多个计算任务可以并发执行，根据任务规模和具体硬件计算能力限制可以并发执行的 Kernel 任务数量可以通过接口 cnGetCtxMaxParallelUnionTask (CNToolkit >= 2.5 或 libcndrv >= 1.5) 获取。

6.5.8.3 内存拷贝和 Kernel 执行并行

数据在设备和主机之间拷贝和 Kernel 的执行可以并行进行。设备内内存拷贝任务和 Kernel 执行也可以并行。例如分别向不同的 CNRT Queue 中下发拷贝和计算任务，由于两个 Queue 中的任务是可以并行执行的，Kernel 计算任务从 Queue 中取出交由 MLU 处理，拷贝任务从 Queue 中取出交由 DMA 处理。但是如果计算和拷贝任务在同一个 Queue 中，由于 Queue 任务是 FIFO (First In First Out, 先进先出) 执行，所以此时两个任务无法并行。

6.5.8.4 内存拷贝任务之间并行

多个设备和主机之间数据拷贝任务可以并行执行。例如分别给设备 A 和设备 B 下发主机侧到设备侧的拷贝任务，对于这两个任务来说是分别使用两个设备的 DMA 来进行。

6.5.9 Queue

Cambricon@155chb

Queue 是用来管理并行操作的一种方式。在同一个 Queue 中所下发的操作顺序是依次按照下发顺序执行的，在不同 Queue 中的执行操作则是乱序的。Queue 所支持的操作不仅包括 Kernel (在设备上的执行程序)，还包括 Notifier、内存拷贝、设备与主机的数据传输等一系列操作，所有需要按序执行的操作都可以下发至同一个 Queue，Queue 在当前操作满足执行条件后会依次按照进入顺序执行操作；而不同 Queue 的操作则不保证顺序，这些操作的顺序是无法预测的。Queue 的查询操作可以查看当前 Queue 的任务执行是否完成，而同步操作在执行成功的时候则可以确保所有的任务执行完成。

6.5.9.1 创建和销毁

通过 cnrtQueueCreate() 接口可以创建 Queue 对象，该对象可以容纳需要进入该 Queue 的所有操作。需要进入 Queue 的操作需要在接口中把创建的 Queue 对象作为参数。创建 Queue 的样例代码如下面代码所示，该样例代码创建了两个 Queue：

```
cnrtQueue_t Queue[2];  
for (int i = 0; i < 2; i++) {  
    if (cnrtQueueCreate(&Queue[i]) != cnrtSuccess) {  
        printf("Create Queue Failed\n");  
    }  
}
```

创建的 Queue 结构体可以作为所有需要 Queue 的接口参数，下面的代码展示了将 Kernel 以及设备与主机数据传输操作放入 Queue 的方法：

```
float *hostAddr;
void *inputAddr;
void *outputAddr;
cnrtHostMalloc(&hostAddr, size * 2);
cnrtMalloc(&inputAddr, size * 2);
cnrtMalloc(&outputAddr, size * 2);

for (int i = 0; i < 2; i++) {
    cnrtMemcpyAsync(inputAddr + i * size, hostAddr + i * size,
                    size, Queue[i], cnrtMemcpyHostToDev);
    Kernel<<<1, 1, 1, Queue[i]>>>(OutputAddr + i * size, inputAddr + i * size, size);
    cnrtMemcpyAsync(hostAddr + i * size, OutputAddr + i * size,
                    size, Queue[i], cnrtMemcpyDevToHost);
}
```

上面的样例代码将输入数据拷贝到设备的拷贝操作放入 Queue[i] 中，并紧接着在同一个 Queue 中放入了 Kernel 操作，再放入了将输出数据传输到主机的拷贝操作，放入同一个 Queue 的这三个操作是依次执行的。而创建的两个 Queue 均放入了三个同样的操作，两个 Queue 中的操作则可以并行执行。两个不同的 Queue 的拷贝操作与 Kernel 执行操作，可以并行执行相互覆盖其中的执行延迟。

Queue 对象可以通过 cnrtQueueDestroy() 接口进行销毁，销毁后该 Queue 对象不可以再用作接口参数进行接收执行操作。

```
for (int i = 0; i < 2; i++) {
    if (cnrtQueueDestroy(Queue[i])) {
        printf("Destroy Queue Failed\n");
    }
}
```

注解：

如果当前 Queue 中有操作未执行或正在执行，调用上面的接口后会立刻返回成功并且销毁还未执行的操作或打断正在执行的操作后，自动释放所有操作所关联的资源。

6.5.9.2 Default Queue

对于 Kernel 的执行操作，其中的 Queue 参数可以不传，或者传入 Queue 参数为 0 时，代表着当前使用默认 Queue。对于所有需要传入 Queue 参数的接口，当传入 Queue 的参数为 0 时，均认为当前使用默认 Queue 进行操作。默认 Queue 与普通的 Queue 功能完全相同，也是对所有操作的并行执行的管理，并且对于每一个设备，每一个主机侧线程都有一个自己的默认 Queue。

注意：

- 用户不能通过接口 `cnrtQueueDestroy()` 进行销毁；
- 默认 Queue 对象的创建与销毁由库自行管理，用户无需管理。

6.5.9.3 显式同步

Queue 的同步操作是用来等待当前 Queue 中的所有任务均执行完成的接口，当前有多种同步的操作。

- `cnrtSyncDevice()` 等待所有设备上的所有 Queue 执行完成。
- `cnrtQueueSync()` 该接口会有一个 Queue 对象参数，接口会等待指定 Queue 的所有操作均执行完成。当 Queue 中的操作发生异常后，该 Queue 会立即返回并返回失败；并且此时 Queue 不再能够下发任务。
- `cnrtQueueQuery()` 该接口会有一个 Queue 对象参数，接口会查询指定 Queue 的操作执行状态；当指定 Queue 中的所有任务均执行完成，则会返回 `cnrtSuccess`，否则返回 `cnrtErrorNotReady`。
- `cnrtQueueWaitNotifier()` 该接口会有一个 Queue 参数以及 Notifier 参数。关于 Notifier 说明请参见 第 6.5.10 章节 Notifier。所有在指定 Queue 中并且在该接口之后下发的操作，均需要等待对应的 Notifier 执行完成。

注意：

`cnrtQueueSync()` 以及 `cnrtQueueQuery()` 接口中的 Queue 参数同样支持传入 0 值，此时则查询默认 Queue 的状态。若 Queue 内部从未有过任何操作，同步接口也会返回成功。

6.5.9.4 同步行为设置

对于 `cnrtSyncDevice()` 以及 `cnrtQueueSync()` 同步操作接口来说，由于可能存在有多个主机侧线程调用同时同步的场景，此时等待的操作同步完成的方式则对性能以及 CPU 的占用率有影响。因此，对于同步操作提供了几种同步的方式，供不同的场景进行选择。

6.5.9.4.1 同步行为的选择方法

可选择的同步行为如下：

```
typedef enum {
    cnrtDeviceScheduleSpin = 0,
    /*!< CPU actively spins when waiting for the Device execution result. */
    cnrtDeviceScheduleBlock = 1,
    /*!< CPU thread is blocked on a synchronization primitive when waiting for the Device
     * execution results. */
    cnrtDeviceScheduleYield = 2,
    /*!< CPU thread yields when waiting for the Device execution results. */
    cnrtDeviceFlagsMaxNum
    /*!< The last one. */
} cnrtDeviceFlags_t;
```

用户在调用同步接口之前，通过 `cnrtSetDeviceFlag()` 接口设置同步操作行为，将对应的同步操作行为枚举 `cnrtDeviceFlags_t` 设置为对应的同步操作行为。

同步操作行为的区别如下：

- `cnrtDeviceScheduleSpin`：调用同步接口的线程，会一直循环查询等待操作完成。此行为性能最优，但在多线程同时调用同步操作的场景下，可能会导致 CPU 占用率较高。一般多用于对性能要求较高，但对 CPU 占用率不敏感的场景。
- `cnrtDeviceScheduleBlock`：调用同步接口的线程，会被阻塞在等待接口中，执行完成后才会继续执行。此行为可以减少多线程调用同步操作时 CPU 占用率，但性能会略微下降。一般多用于对性能不敏感，但要求 CPU 占用率较低的场景。
- `cnrtDeviceScheduleYield`：调用同步接口的线程，会被阻塞在等待接口中，并主动让出阻塞线程供其他执行线程进行执行。此行为主动让出当前线程，因此当前 CPU 负载较高时其性能会较低，而负载较低时性能较高；仅有同步操作线程时，其行为等同于 `cnrtDeviceScheduleSpin`。

注意：

对于端侧平台，当前默认方式为 `cnrtDeviceScheduleBlock`，其主要目的是为了最大化利用端侧 CPU 资源；而对于云侧平台，默认为 `cnrtDeviceScheduleSpin` 方式，主要是为了最大化性能。

6.5.9.5 不同 Queue 操作的重叠行为

由于同一个 Queue 操作顺序执行，不同 Queue 则乱序执行，因此，对于不同 Queue 的大量操作，会出现重叠行为 (Overlap Behavior)。

对于下面的用例代码：

```
for (int i = 0; i < 2; i++) {
    cnrtMemcpyAsync(inputAddr + i * size, hostAddr + i * size,
                    size, Queue[i], cnrtMemcpyHostToDev);
    Kernel<<<1, 1, 1, Queue[i]>>>(OutputAddr + i * size, inputAddr + i * size, size);
    cnrtMemcpyAsync(hostAddr + i * size, OutputAddr + i * size,
                    size, Queue[i], cnrtMemcpyDevToHost);
}
```

由于主机与设备的数据拷贝操作可以与 Kernel 并行执行，因此对于 Queue[1] 上的主机到设备的数据拷贝操作就与 Queue[0] 上的 Kernel 重叠；同样而言，Queue[1] 上的 Kernel Launch 则与 Queue[0] 上的设备到主机数据拷贝操作重叠。该重叠行为可以掩盖掉一部分执行时间，从而提高代码性能。

6.5.10 Notifier

Notifier 是一种特殊的任务，和计算任务一样可以放置到队列中执行，Notifier 完成条件是 Queue 中前序任务都完成，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，可以通过两个 Notifier 来实现对计算任务耗时的精确统计。此外应用程序可以在执行过程中通过异步 Notifier API 完成在 Queue 中放置 Notifier，然后通过查询 Notifier 完成状态来判断 Queue 中任务完成状态。

表 6.4: CNRT 常用 Notifier 管理接口

API 名称	功能简介
cnrtNotifierCreate()	创建一个 Notifier。
cnrtNotifierDestroy()	删除一个 Notifier。
cnrtPlaceNotifier()	将 Notifier 放到一个任务队列中。
cnrtNotifierElapsedTime()	计算两个 Notifier 之间所消耗的时间（单位：微秒）。
cnrtQueueWaitNotifier()	任务队列等待一个 Notifier 完成（用于多队列间同步）。

6.5.10.1 创建和销毁 Notifier

创建和销毁两个 Notifier 示例代码示例如下。

```
cnrtNotifier_t notifier_start;
cnrtNotifier notifier_end;

/* create notifier*/
ERROR_CHECK(cnrtNotifierCreate(&notifier_start));
/* create notifier*/
ERROR_CHECK(cnrtNotifierCreate(&notifier_end));

/* Destroy notifier*/
ERROR_CHECK(cnrtQueueDestroy(notifier_start));
/* Destroy notifier*/
ERROR_CHECK(cnrtQueueDestroy(notifier_end));
```

6.5.10.2 统计时间

使用 Notifier 统计时间的代码示例如下。

```
Cambricon@155chb
void cnrtNotifierDurationKernel(void) {
    cnrtNotifier_t st, et[5];
    cnrtQueue_t queue;

    ERROR_CHECK(cnrtNotifierCreate(&st));
    for (int i = 0; i < 5; i++) {
        ERROR_CHECK(cnrtNotifierCreate(&et[i]));
    }

    ERROR_CHECK(cnrtQueueCreate(&queue));

    ERROR_CHECK(cnrtPlaceNotifier(st, queue));
    for (int i = 0; i < 5; i++) {
        simpleKernelAdd(queue);
        ERROR_CHECK(cnrtPlaceNotifier(et[i], queue));
    }

    ERROR_CHECK(cnrtQueueSync(queue), cnrtSuccess);
    float ms[5];
    for (int i = 0; i < 5; i++) {
        ERROR_CHECK(cnrtNotifierDuration(st, et[i], &ms[i]));
    }
}
```

```
    printf("basic test duration is %f\n", ms[i]);  
}  
  
    ERROR_CHECK(cnrtNotifierDestroy(st));  
    for (int i = 0; i < 5; i++) {  
        ERROR_CHECK(cnrtNotifierDestroy(et[i]));  
    }  
  
    ERROR_CHECK(cnrtQueueDestroy(queue));  
    ERROR_CHECK(cnrtDeviceReset());  
}
```

6.5.10.3 多队列间同步

用 Notifier 实现多队列间同步的代码示例如下：

```
// set: dev_mem0, dev_mem1, host_mem0, host_mem1, size...  
cnrtNotifier_t notifier_0;  
cnrtNotifier_t notifier_1;  
cnrtNotifierCreate(&notifier_0);  
cnrtNotifierCreate(&notifier_1);  
  
cnrtQueue_t queue_0;  
cnrtQueue_t queue_1;  
cnrtQueueCreate(&queue_0);  
cnrtQueueCreate(&queue_1);  
  
cnrtMemcpyAsync(dev_mem0, host_mem0, size, queue_0, cnrtMemcpyHostToDev);  
cnrtPlaceNotifier(notifier_0, queue_0);  
cnrtMemcpyAsync(host_mem1, dev_mem1, size, queue_1, cnrtMemcpyDevToHost);  
cnrtPlaceNotifier(notifier_1, queue_1);  
  
// queue_1 wait queue_0, queue_0 wait queue_1  
cnrtQueueWaitNotifier(notifier_0, queue_1, 0);  
cnrtQueueWaitNotifier(notifier_1, queue_0, 0);  
  
cnrtQueueSync(queue_0);  
cnrtQueueSync(queue_1);
```

6.5.11 错误检查

CNRT 提供了一系列的错误管理接口，帮助用户更方便地调试 CNRT 程序。

表 6.5: CNRT 常用错误管理接口

API 名称	功能简介
cnrtGetErrorName()	获取错误码表示的错误名。
cnrtGetErrorStr()	获取错误码表示的错误信息。
cnrtGetLastError()	获取最近一次调用 CNRT 接口的错误码，错误码返回后会重置为 cnrtSuccess。
cnrtPeekAtLastError()	获取最近一次调用 CNRT 接口的错误码，错误码返回后不会重置。

6.6 软件版本兼容性

CNCC 和 CNRT 都是 CNToolkit 的组件，两者的版本各自遵循向后兼容的原则，即新版本兼容老版本，但是两者的版本不能任意组合使用，需要使用 CNToolkit 发布版中的版本组合，详见《寒武纪 CNToolkit 安装升级使用手册》。

在开发 Cambricon BANG C 程序时需要关注两个版本号，一种是 Cambricon BANG 异构并行计算平台的版本号，一种是 CNToolkit 的版本号。前者决定了 Cambricon BANG 异构计算平台能够向上层提供的硬件特性和计算能力，而后者决定了程序员在当前 CNToolkit 版本能够使用的硬件特性和软件功能。如表 6.6 Cambricon BANG 异构计算平台版本与 CNToolkit 版本的对应关系 所示。

表 6.6: Cambricon BANG 异构计算平台版本与 CNToolkit 版本的
对应关系

Cambricon BANG 版本号	CNToolkit 版本号	MLISA 版本号	CNCC 版本号	架构代号
Cambricon BANG v4.0.x	CNToolkit-v4.0.x	MLISA-V4.0.x	CNCC-v4.0.x	mtp_592, tp_322, mtp_372, mtp_290, (m)tp_270, (m)tp_220, mtp_100, 1H8, 1H8 mini
Cambricon BANG v3.3.x	CNToolkit-v2.3.x	MLISA-v3.3.x	CNCC-v3.3.x	tp_322, mtp_372, mtp_290, (m)tp_270, (m)tp_220, mtp_100, 1H8, 1H8 mini
Cambricon BANG v3.0.x	CNToolkit-v2.0.x	MLISA-v3.0.x	CNCC-v3.0.x	mtp_372, mtp_290, (m)tp_270, (m)tp_220, mtp_100, 1H8, 1H8 mini
Cambricon BANG v2.15.x	CNToolkit-v1.7.x	MLISA-v2.15.x	CNCC-v2.15.x	mtp_290, (m)tp_270, (m)tp_220, mtp_100, 1H8, 1H8 mini



7 性能调优指南

本章先介绍性能调优的总体原则，再给出性能调优的两个大方向：计算效率最大化和 IO 效率最大化。

7.1 性能调优的总体原则与流程

本节先介绍两个定律，再给出性能调优的基本原则和基本流程。

7.1.1 阿姆达尔定律

阿姆达尔定律 (Amdahl's Law) 是一个并行处理性能模型，其描述了并行计算或并行存储系统中，程序并行化可以获得的加速比。公式如下：

$$S = \frac{1}{1-P+\frac{P}{N}}$$

Cambricon@155chb

其中， P 代表程序中并行部分的比例，是个 $[0, 1]$ 之间的实数， $1-P$ 代表程序中串行部分的比例，取值也在 $[0, 1]$ 之间， N 指并行处理节点个数， S 指整个程序相对于串行程序的加速比。

例如，当一个程序全是并行代码时，并行部分比例 $P=1$ ，加速比 $S=N$ ，即加速比等于并行处理节点个数，相比于单处理节点的串行代码，加速了 N 倍。再如，当一个程序全是串行代码时，并行部分比例 $P=0$ ，加速比 $S=1$ ，即相比于单处理节点的串行代码没有加速。特别地，当 N 趋近于无穷大时，上述公式近似为 $S = \frac{1}{1-P}$ ，这代表了加速比的上限。比如当一个程序并行代码比例 $P=0.8$ 时，加速比 $S=5$ ，即并行代码的性能不会超过串行代码性能的 5 倍。

该定律说明，如果一段程序中并行代码的比例 P 较小，那么仅靠增加处理器数量 N 也不会获得较大的性能提升。若要取得较大加速比，需要用户充分挖掘自己程序中可以并行的部分，提升并行代码占比。

7.1.2 古斯塔夫森定律

古斯塔夫森定律 (Gustafson's Law) 是阿姆达尔定律的延伸，公式如下：

$$S = N - F(N - 1)$$

其中， F 代表程序中串行部分的比例，是个 $[0, 1]$ 之间的实数， $1-F$ 即程序中并行部分的比例（和阿姆达尔定律中的 P 相同）， N 代表并行处理节点个数， S 指整个程序相对于串行程序的加速比。

该定律说明，当一段程序中并行代码的比例较高时（ F 接近 0），加速比和并行处理节点数成正比。即增加处理器核数是有必要的，但前提是程序中并行部分占比高。

7.1.3 性能调优基本原则

通过前面的两个定律可知，在问题规模和硬件参数固定的条件下，性能调优最重要的方法是提高程序的并行度。针对 MLU 提供的并行能力，可以从计算并行、计算与 IO 并行两个方向提高程序的并行度。

需要注意的是，性能调优要结合具体硬件的差异，在一款 MLU 上性能最好的代码，放到另一款 MLU 上性能未必最佳。

7.1.4 性能调优基本流程

1. 建立性能基准：以第一份运行正确的代码运行时间为基准性能数据。
2. 识别性能瓶颈：程序性能瓶颈主要有两个，计算瓶颈和 IO 瓶颈。
3. 等价变换：确保代码执行正确的前提下，优化代码，消除或降低瓶颈。
4. 回归验证：运行新代码，看性能是否提升，然后继续上述步骤，不断优化。

需要注意的是，性能调优是一个不断迭代的过程，每一轮迭代应将优化重点放在性能瓶颈上（计算瓶颈或 IO 瓶颈）。

Cambricon@155chb

7.2 计算效率最大化

计算效率最大化是指通过优化计算指令，尽可能提高程序对硬件各运算器的使用率，降低计算耗时。

具体来说有三类常用方法：计算并行，减少计算量和等效替代。

7.2.1 计算并行

基于 MLU 硬件的并行计算系统支持 7 个层次的并行计算：服务器级别并行、板卡级别并行、芯片级别并行、Cluster 级别并行、MLU Core 级别并行、指令级并行和数据级并行。

计算效率最大化就是将算法映射到不同的并行层次的过程。其中，服务器级别并行、板卡级别并行一般面向人工智能训练和大规模推理任务，通常借助 CNCL 通信库和 CNRT 运行时库完成。而其他层级的并行计算能力则是可以通过 Cambricon BANG C 语言编程实现。

7.2.2 减少计算量

从源头减少计算量一直是性能提升的通用手段，比如算法优化、删除加 0 或者乘 1 操作、宏代替常量表达式、常数预处理等。其中常数预处理是指，对于不随输入数据变化而变化的常量数据，可以在 CPU 上计算完成后，传入 MLU GDRAM，以便 MLU 直接读取，减少 MLU 计算量。

7.2.3 等效替代

等效替换是指用运算速度较快的等价操作替换运算速度较慢的操作。比如，计算整数 b 乘以 8，`b * 8` 和 `b << 3` 相比，后者的性能更高。

7.3 IO 效率最大化

IO 效率最大化是指通过优化访存指令的行为，尽可能提高程序对硬件访存部件的使用率，降低访存耗时，尽可能实现访存与计算的并行。访存优化有三类常用方法：减少访存量，计算和访存并行，提升带宽利用率。

Cambricon@155chb



8 性能调优实践

本章首先介绍性能分析的基本手段，然后介绍计算和访存优化的基本方法，最后再以一个具体的案例来展示各种优化方法的综合运用。

8.1 性能分析的基本手段

8.1.1 CNPerf

CNPerf (Cambricon Performance, 寒武纪性能剖析工具) 是一款面向 Cambricon BANG 异构并行编程模型的性能剖析工具，可以用于主机侧与设备侧并行度进行调优，也可以用于单 Kernel 进行调优。

主机侧与设备侧并行度调优，即端到端调优（下文简称为 E2E 调优），主要利用 CNPerf 获取主机侧与设备侧的执行流 (timeline/timechart)。涉及 CNPerf 的 record、report、timechart 等命令。

在单 Kernel 调优的过程中，主要利用 CNPerf 获取设备侧的 PMU (Performance Monitoring Unit，性能监控单元) 性能数据来分析 Kernel 的性能。涉及 CNPerf 的 record、monitor、kernel、parse 等命令。

注意：

- 使用 CNPerf 时请保证 MLU 设备无预期外的任务在运行，否则获取到的性能数据不可靠。
- 不同 MLU 设备间存在性能事件类型及数目差异，运行 CNPerf 时，该 MLU 设备若无此性能事件，则显示 N/A 或不显示。
- 不同版本的 CNPerf，各命令使用方式及显示效果可能有差别，具体以《寒武纪 CNPerf 用户手册》为准。

8.1.1.1 CNPerf E2E 调优

在 Cambricon BANG 异构并行编程模型中，CPU 和 MLU 协同完成计算，用户在 E2E 调优过程中，首要目标就是确保在关键的计算路径上，MLU 不存在饥饿、空闲、调度阻塞等情况，其次，用户需要识别热点 MLU 算子，以挖掘较大的优化空间。

本文以 TensorFlow 的 BERT NER 网络作为被测例，简述 CNPerf E2E 调优相关的基本功能以及性能分析方法。

在正常情况下，用户使用如下命令执行被测例：

```
python BERT_NER.py (省略参数...)
```

使用 CNPerf 进行调优时，无需修改原程序，只需修改执行命令，如下所示：

```
cnperf-cli record --args python BERT_NER.py (省略参数...)
```

执行成功后，使用如下命令生成 timechart 日志文件：

```
cnperf-cli timechart
```

执行成功后，在当前执行目录下，该命令会生成一个 timechart_data.json 文件。

使用 Chrome 浏览器，在网址栏中输入 chrome://tracing/，点击 Load 按钮，打开刚刚生成的 timechart_data.json 文件，如 图 8.1 tracing 默认界面 所示：



图 8.1: tracing 默认界面

打开日志文件后，如 图 8.2 timechart 可视化界面 所示：



图 8.2: timechart 可视化界面

该界面是用户查看主机侧及设备侧执行流的主要操作界面：

- 红框 1：各种执行流的名称，包括主机侧执行流对应的线程号，以及设备侧执行流对应的设备号及 Queue ID。
- 红框 2：提供了一些界面的显示控制功能，本文中主要使用其中的“Flow events”功能。
- 红框 3：操作模式选择功能，主要影响鼠标的操作模式，本文主要使用最上方的模式 1，即选择模式。
- 红框 4：提供了关闭已经显示的执行流的功能，当用户不关心特定的执行流时，可使用该功能关闭对应的执行流。
- 红框 5：执行流的主要显示区域。

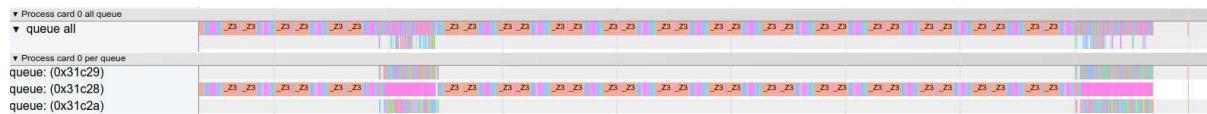


图 8.3: timechart 中的设备侧执行流界面

CNPerf timechart 可视化界面中同时为用户提供了主机侧和设备侧的执行流：

- 主机侧执行流（图 8.2 timechart 可视化界面 中以 Thread 开头的执行流）包括寒武纪软件栈相关 API 等信息；
- 设备侧执行流（图 8.3 timechart 中的设备侧执行流界面 中的“Process card 0 per/all queue”）包括用户程序下发的 Kernel、算子以及异步内存拷贝等其它设备侧相关的异步任务操作。

注解：

下文中提及的与设备侧执行流相关的 Kernel、算子、任务，均指通过 CNDrv、CNRT 的 Queue 相关接口下发到设备上执行的异步任务。

使用 CNPerf 进行的 E2E 调优，主要围绕设备侧执行流展开，包括以下两个重点内容：

1. 寻找设备侧执行流中的热点算子，即设备侧执行流中累计执行时间较长的算子。对于总耗时越大的算子，优化获得的潜在收益越大。
2. 寻找设备侧执行流中的空泡，即算子与算子之间的空白。空泡代表着 MLU 处于饥饿、空闲、调度阻塞等状态，MLU 算力得不到充分发挥，用户需要进一步分析空泡产生的原因。

8.1.1.1 寻找设备侧执行流中的热点算子

通过缩放调整到第二次迭代附近，在选择模式（模式 1）下，在空白处按住鼠标左键拉出一个选择框，将设备侧执行流中的所有 Kernel 都选进去（queue 0x31c28 那一行），选的时候不要包含其它 Queue 中的事件。



图 8.4: 选中设备侧执行流中 Kernel 的示意图

选定后下面的对话框会给出这一段设备侧执行流的汇总信息，从汇总信息中可以看到所有算子的总执行时间（Wall Duration）、平均执行时间（Average Wall Duration）以及出现次数（Occurrences）。

点击“Wall Duration”，按照总执行时间排序。

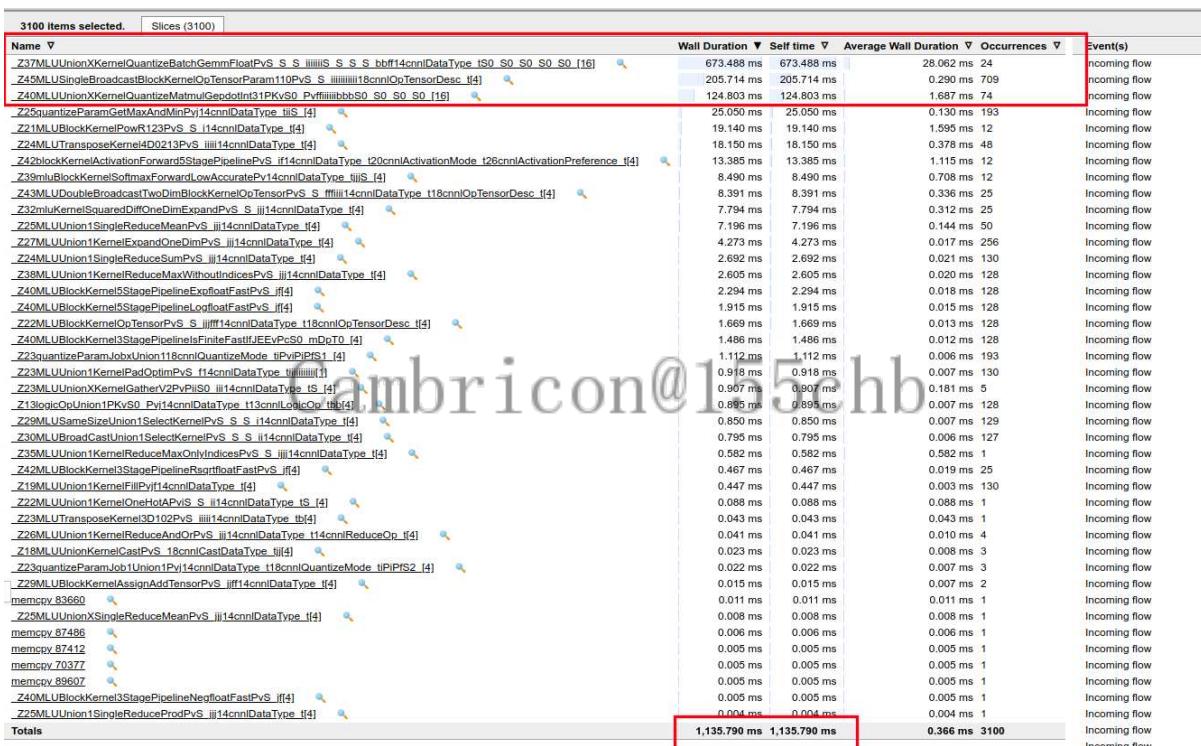


图 8.5: Wall Duration 排序显示效果

如图 8.5 Wall Duration 排序显示效果所示，累计执行时间最长的 3 个 Kernel 的总执行时间占比为： $(673.5 + 205.7 + 124.9) / 1135.8 = 88.4\%$ 已经占到了总体硬件时间的 88.4%，属于明显的热点算子。

针对热点算子去做优化，可以获得显著的性能收益。

注解：

- 表格下方的 Totals 时间是所有算子的硬件执行时间，不包含空泡的时间。
- 若用户使用的是寒武纪提供的算子库，例如 CNNL（寒武纪人工智能计算库），则可以根据相关用户手册中的调优建议，结合实际情况后对业务逻辑作出调整，或联系寒武纪工程师沟通相关事宜。

8.1.1.1.2 寻找设备侧执行流中的空泡



图 8.6: 主机侧与设备侧执行流的整体状况

如图 8.6 主机侧与设备侧执行流的整体状况 所示，从全局图形上可以看到网络前后两段（红框 vs 蓝框）的行为模式上有明显差异。

前半段（上图红框），设备侧执行流中有许多执行时间较长的算子，且算子排列比较紧密。

将界面放大到后半段附近（上图蓝框），可以看到设备侧执行流中的算子比较细碎，并且有明显的空泡。

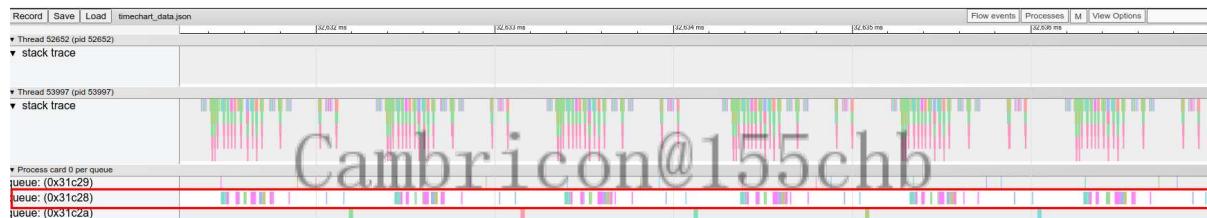


图 8.7: 设备侧执行流中空泡较为明显的区域

造成这种空泡的原因，最常见的有 2 种：

1. 主机侧算子在计算图上没有依赖，就是单纯的下发 Kernel 的速度不够快。
2. 主机侧算子在计算图上有依赖，需要等前序依赖执行完毕才能下发后续的 Kernel。

用户可以通过打开右上角的“Flow events”来辅助区分识别这两种状态：



图 8.8: 操作示意图：打开 Flow event

主机侧算子在计算图上没有依赖，单纯的下发 Kernel 不够快

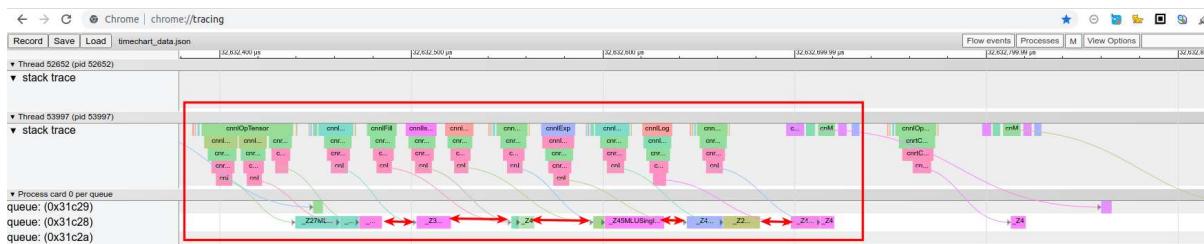


图 8.9: 计算图没有依赖的示意图

打开“Flow events”后，可以可视化地看到主机侧下发 Kernel 到 Kernel 实际在硬件的计算队列中从开始到结束的完整过程。

从上面这张图可以看到，主机侧（Thread 53997）执行比较紧密，基本没有空泡（微小的空白是正常的软件开销），主机侧运转是顺畅的，但是设备侧产生了明显的空泡。

空泡原因比较直观，因为设备侧运行的 Kernel 都很短，只有几微妙或者十几微妙，设备侧硬件时间比主机侧下发 Kernel 所需的时间还短，直接导致了 MLU 饥饿。

用户应当确保 Kernel 的执行时间大于下发 Kernel 所需的时间，以下是常见做法：

- 尝试调整单个 Kernel 的运算规模，例如调整 batch size 等。
- 尝试融合多个小 Kernel 为一个大 Kernel。

主机侧算子在计算图上有依赖，需要等前序依赖执行完毕才能下发后续的 Kernel



图 8.10: 计算图有依赖的示意图

在上述样例中，可以明显看到红色的 1 号标记位置，主机侧 CPU 线程由于某些原因，没有连续下发 Kernel。

用户应当具体分析主机侧线程空泡处的程序行为，尽可能避免影响任务下发的操作。

小技巧：

用户可以根据 Flow event 的倾斜程度，大体判断主机侧执行流下发任务是否存在瓶颈。

如图 8.11 Flow event 倾斜程度示意图 所示，右侧 Flow event 倾斜度较大，且设备侧执行流中的 Kernel 较为连续，表示主机侧下发任务基本不是瓶颈。在这种情况下，下一步的调优目标应当是优先减少 Kernel 的总执行时间，即寻找设备侧执行流中的热点算子。

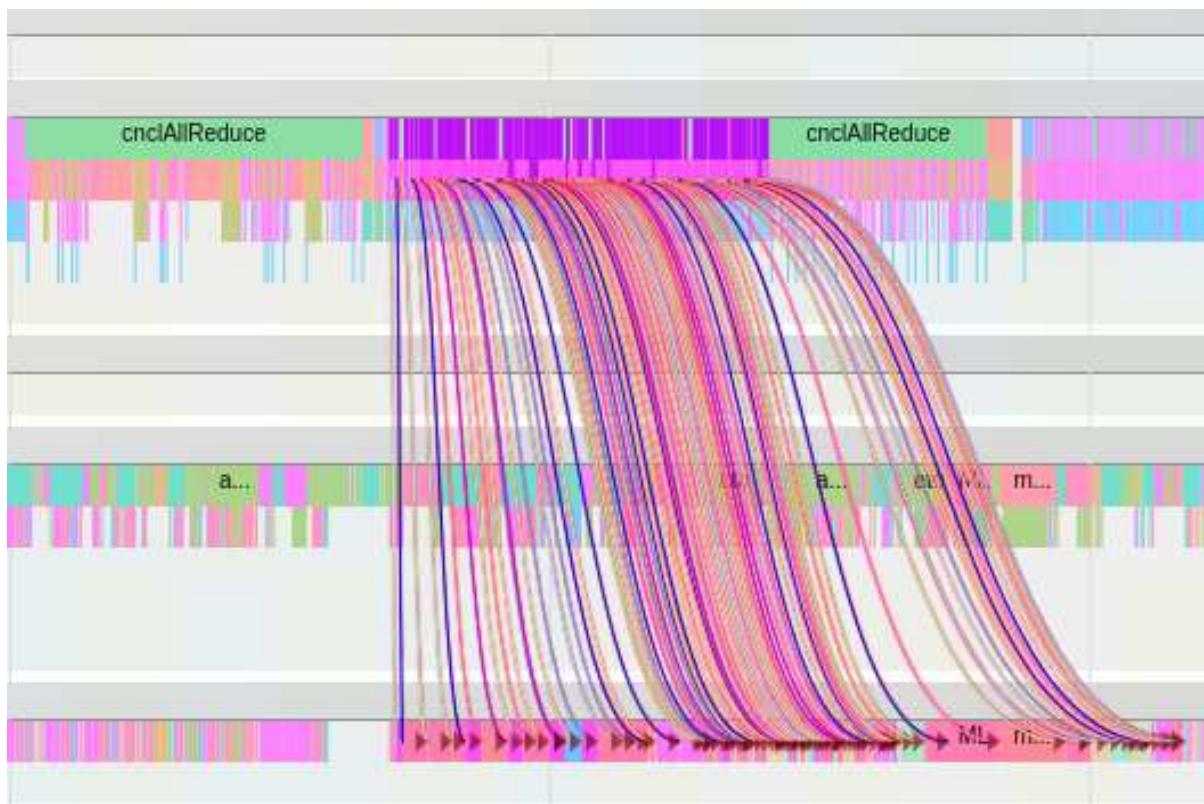


图 8.11: Flow event 倾斜程度示意图

注解:

Cambricon@155chb

- chrome://tracing/ 右上角的“?”按键提供了界面操作相关的帮助信息，用户可根据帮助信息熟悉界面操作。
- 不同版本的 CNPerf，提供的 timechart 信息及其显示效果可能会有差异，具体以《寒武纪 CNPerf 用户手册》为准。

8.1.1.2 cnperf-cli monitor 命令

monitor 命令用于查看 MLU 工作时的性能指标，如读写带宽等。

注解:

PMU 性能事件的具体含义，详见《寒武纪 CNPerf 用户手册》。

8.1.1.2.1 使用方法

在任意目录执行命令 cnperf-cli monitor 即可。

注解：

- 在 Cambricon BANG C 性能分析中，monitor 命令用于旁路查看硬件性能指标。由于 kernel 一般执行时间较短，所以可以先运行 monitor，然后再运行 kernel，等 kernel 运行结束，停止 monitor。这时查看 monitor 命令输出的数据来获得 Kernel 运行的性能指标。
- monitor 命令也可以用于在混合执行业务时，查询设备带宽信息，例如查看 vpu、tp_core 等模块的读写带宽。

8.1.1.2.2 显示效果

```
CNPerf Monitor:  
execution time(ns) : [0s, 589ms, 839us, 826ns]  
  
Unit: mlu  
-----  
power: W 27  
temperature: C 32  
cpu_utils: % 25  
  
Unit: tp_cluster  
-----  
write_bytes: MB 0.000  
write_bw: MB/s 0.000  
read_bytes: MB 0.000  
read_bw: MB/s 0.000  
bandwidth_utils: % 0.000  
  
Unit: tp_core  
-----  
write_bytes: MB 0.000  
write_bw: MB/s 0.000  
read_bytes: MB 0.000  
read_bw: MB/s 0.000  
tlb_write_access: times 0  
tlb_read_access: times 0
```

```
tlb_write_miss:           times          0
tlb_read_miss:           times          0
core_utils:               %              0
lt_cycles:                cycles         0
lt_utils:                 %             0.000
alu_cycles:                cycles         0
alu_utils:                 %             0.000
csimd_post_cycles:       cycles         0
csimd_post_utils:        %             0.000
```

...

8.1.1.3 通过 cnperf-cli record --pmu 命令进行单算子调优

通过 record 命令添加 --pmu 参数启动程序，获取程序运行中的数据并记录到日志文件。然后通过 kernel 命令解析生成的文件，获取 Kernel 性能数据。

8.1.1.3.1 使用方法

Cambricon@155chb

- 运行

使用命令跟踪可执行程序 cnperf-cli record --pmu <USER BINARY>。

- 解析

运行 kernel 命令解析日志 cnperf-cli kernel。

注意：

- 使用 --pmu 参数时，CNPerf 会将所有的 Kernel 任务下发行改为同步执行。
- 若用户在程序中创建了多个子进程，CNPerf 只会提供主进程的 Kernel 性能数据。
- record 命令相关的注意事项，详见《寒武纪 CNPerf 用户手册》。

8.1.1.3.2 显示效果

```
Kernels Info:
=====
Kernel Name  : add1[BLOCK]
Dim          : 1 * 1 * 1
TID          : 22794
```

```
Duration      : [0s, 887ms, 518us, 756ns]
```

```
Device ID : 0
```

```
Unit: tp_cluster
```

```
write_bytes:          bytes      67108864
write_bw:            MB/s       75.614
read_bytes:          bytes     134226176
read_bw:            MB/s      151.238
bandwidth_utils:    %          0.074
```

```
Unit: tp_core
```

```
write_bytes:          bytes      0
write_bw:            MB/s       0.000
read_bytes:          bytes      0
read_bw:            MB/s       0.000
tlb_write_access:   times     1048576
tlb_read_access:   times     2097169
tlb_write_miss:    times      1
tlb_read_miss:    times      24
lt_cycles:          cycles     0
lt_utils:           %          0.000
alu_cycles:          cycles     0
alu_utils:           %          0.000
csimd_post_cycles: cycles     0
csimd_post_utils:  %          0.000
```

Cambricon@155chb

8.1.1.4 通过 cnperf-cli record --kernel_profiling 命令进行单算子调优

使用 CNCC 编译时，用户应增加 -fbang-instrument-kernels 参数，以使能编译器对源函数进行插桩处理。

通过 cnperf record 命令添加 --kernel_profiling 参数启动程序，获取程序运行中的数据并记录到日志文件。通过 cnperf parse 命令解析生成的日志，将 Kernel 性能数据处理成 csv 文件。

注意：

- `-fbang-instrument-kernels` 只会对 `__mlu_entry__` 修饰的函数进行插桩。
- 在编译时，编译器不会对 `__mlu_func__` 修饰的函数进行插桩处理。

8.1.1.4.1 使用样例

- 源代码 test.mlu

```
#include <bang.h>

#define ELEM_NUM 128

__mlu_entry__ void kernel(float* dst, float* src0, float* src1) {
    __nram__ float dst_nram[ELEM_NUM];
    __nram__ float src0_nram[ELEM_NUM];
    __nram__ float src1_nram[ELEM_NUM];
    __memcpy(src0_nram, src0, ELEM_NUM * sizeof(float), GDRAM2NRAM);
    __memcpy(src1_nram, src1, ELEM_NUM * sizeof(float), GDRAM2NRAM);
    __bang_add(dst_nram, src0_nram, src1_nram, ELEM_NUM);
    __memcpy(dst, dst_nram, ELEM_NUM * sizeof(float), NRAM2GDRAM);
    return;
}

int main() {
    CNRT_CHECK(cnrtSetDevice(0));
    cnrtQueue_t queue;
    CNRT_CHECK(cnrtQueueCreate(&queue));
    cnrtDim3_t dim = {1, 1, 1};
    cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_BLOCK;

    float src_cpu[ELEM_NUM];
    float dst_cpu[ELEM_NUM];
    for (unsigned i = 0; i < ELEM_NUM; ++i) {
        src_cpu[i] = 1.0f;
    }

    float* src_mlu = NULL;
    float* dst_mlu = NULL;
    CNRT_CHECK(cnrtMalloc((void**)&src_mlu, ELEM_NUM * sizeof(float)));
    CNRT_CHECK(cnrtMalloc((void**)&dst_mlu, ELEM_NUM * sizeof(float)));
    CNRT_CHECK(cnrtMemcpy(src_mlu, src_cpu, ELEM_NUM * sizeof(float),
        CNRT_MEM_TRANS_DIR_HOST2DEV));
```

```

kernel<<<dim, func_type, queue>>>(dst_mlu, src_mlu, src_mlu);
CNRT_CHECK(cnrtQueueSync(queue));
CNRT_CHECK(cnrtMemcpy(dst_cpu, dst_mlu, ELEM_NUM * sizeof(float),
                      CNRT_MEM_TRANS_DIR_DEV2HOST));
CNRT_CHECK(cnrtFree(src_mlu));
CNRT_CHECK(cnrtFree(dst_mlu));
CNRT_CHECK(cnrtQueueDestroy(queue));

// 1.0f + 1.0f = 2.0f
for (unsigned i = 0; i < ELEM_NUM; ++i) {
    printf("dst[%u] %f\n", i, dst_cpu[i]);
}

return 0;
}

```

- 编译

以 mtp_372 架构为例，编译命令：

```
cncc test.mlu --bang-mlu-arch=mtp_372 -o test -fbang-instrument-kernels
```

- 运行

通过如下命令运行：

```
cnperf-cli record --kernel_profiling ./test
```

在当前目录下会生成文件夹 dltrace_data，用于存放获取的性能数据。

- 解析

在当前目录下运行 cnperf-cli parse 命令，生成文件 cnperf_ipu_perf_data.csv 和 cnperf_mpu_perf_data.csv，分别对应 MLU Core 和 Memory Core 的性能数据。当任务类型为 BLOCK 时，cnperf_mpu_perf_data.csv 文件数据无效。

以 test.mlu 为例，执行 cnperf-cli parse 命令后，cnperf_ipu_perf_data.csv 中包含的信息如下所示：

```

// device_id,tid,perf_cnt,function name,task_id,duration(ns),inst_executed,inst_cache_miss,
→scalar_inst_executed,alu_cycles,read_bytes,write_bytes,csimd_pre_cycles,csimd_post_cycles,
→lt_cycles,ctrampio_inst_read_bytes,ctrampio_inst_write_bytes,ctrampmv_inst_read_bytes,
→ctrampmv_inst_write_bytes,ltram_multicast_bytes,ltram_unicast_bytes,
0,22279,0,"kernel(float*, float*, float*)",0,22160,276,8,90,259,5184,3328,0,28,0,768,1024,0,
→0,0,0,

```

注解：

- device_id：当前设备 ID；
- tid：线程 ID；
- perf_cnt：线程相关计数器，用于区分同一线程不同 Kernel launch；
- function_name：设备侧 Kernel 名；
- task_id：任务 ID。

注意：

- 插桩本身也包含一定开销，会对 IO 数据量、执行指令条数以及 cache miss 等产生细微的影响。
- 该功能不支持与后文的 gettimeofday 功能混用。
- 该功能从 CNToolkit 2.5 之后支持。
- 该功能仅支持 MLUv03 及以上架构。
- 该功能仅支持使用 <<<...>>> 形式启动 Kernel。

8.1.1.5 小结

1. record 命令配合 report 或 timechart 命令，可以显示主机侧与设备侧的工作状态，以便于用户进行 E2E 调优，提高设备侧利用率。
2. monitor 命令使用比较简单，启动 monitor，运行程序，程序运行结束关闭 monitor 即可获取 Kernel 执行的性能数据。该命令也可以用于查看设备侧整体的工作状态及带宽信息。
3. record --pmu 与 kernel 命令可以将 PMU 数据对应到每个 Kernel 上，并提供了 Kernel 执行时间等信息。
4. record --kernel_profiling 与 parse 命令可以将 PMU 数据对应到每个 Task 上，但使用较为繁琐。

8.1.2 gettimeofday

Cambricon BANG C 用户可以通过在设备侧调用 gettimeofday 接口获取代码片段的运行时间，此时间仅包含 gettimeofday 之间设备侧代码片段的时间，设备侧示例代码如下所示：

```
#include <bang.h>
#include <sys/time.h>
#define ELEM_NUM 128

__mlu_entry__ void kernel(float* dst, float* src0, float* src1) {
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    __nram__ float dst_nram[ELEM_NUM];
    __nram__ float src0_nram[ELEM_NUM];
    __nram__ float src1_nram[ELEM_NUM];
```

```
__memcpy(src0_nram, src0, ELEM_NUM * sizeof(float), GDRAM2NRAM);
__memcpy(src1_nram, src1, ELEM_NUM * sizeof(float), GDRAM2NRAM);
__bang_add(dst_nram, src0_nram, src1_nram, ELEM_NUM);
__memcpy(dst, dst_nram, ELEM_NUM * sizeof(float), NRAM2GDRAM);
gettimeofday(&end, NULL);
__bang_printf("Time consumed (device): %fus\n",
              (float)((end.tv_sec - begin.tv_sec) * 1000000
              + (end.tv_usec - begin.tv_usec)));
return;
}
```

8.1.3 Notifier

Cambricon BANG C 用户可以通过在主机侧调用运行时 Notifier 相关接口获取 Kernel 运行时间，此时间包括驱动启动、编译器在用户代码前插入的初始化代码、用户代码等，示例代码如下所示（主机侧）：

```
cnrtNotifier_t begin, end;
float time = 0.0f;
CNRT_CHECK(cnrtNotifierCreate(&begin));
CNRT_CHECK(cnrtNotifierCreate(&end));
CNRT_CHECK(cnrtPlaceNotifier(begin, queue));
kernel<<<dim, func_type, queue>>>(dst_mlu, src_mlu, src_mlu);
CNRT_CHECK(cnrtPlaceNotifier(end, queue));
CNRT_CHECK(cnrtQueueSync(queue));
CNRT_CHECK(cnrtMemcpy(dst_cpu, dst_mlu, ELEM_NUM * sizeof(float),
                      CNRT_MEM_TRANS_DIR_DEV2HOST));
CNRT_CHECK(cnrtNotifierDuration(begin, end, &time));
printf("Time consumed (host): %fus\n", time);
```

8.2 计算效率最优化

8.2.1 使用向量接口优化性能

在 Cambricon BANG C 程序的优化方法中，效果最明显的就是充分利用 Cambricon BANG C 提供的向量接口。下面给出一些使用 Cambricon BANG C 向量接口实现性能优化的经典实例。有关 Cambricon BANG C 向量接口的详细信息可以参考《Cambricon BANG C Developer Guide》。

8.2.1.1 向量相加

- 标量版

```
#define CHANNELS 8192  
...  
for (int i = 0; i < CHANNELS; i++) {  
    dst_nram[i] = src0_nram[i] + src1_nram[i];  
}
```

- 向量版

```
#define CHANNELS 8192  
...  
__bang_add(dst_nram, src0_nram, src1_nram, CHANNELS);
```

向量减法等操作同理。

8.2.1.2 relu 激活

- 标量版

```
#define CHANNELS 8192  
...  
for (int i = 0; i < CHANNELS; i++) {  
    dst_nram[i] = src_nram[i] > 0 ? src_nram[i] : 0;  
}
```

- 向量版

```
#define CHANNELS 8192  
...  
__bang_active_relu(dst_nram, src_nram, CHANNELS);
```

8.2.1.3 对位最大值

- 标量版

```
#define LEN 1024  
...  
for (int i = 0; i < LEN, ++i) {  
    dst[i] = src0[i] > src1[i] ? src0[0] : src1[i];  
}
```

- 向量版

```
#define LEN 1024  
...  
__bang_maxequal(dst, src0, src1, LEN);
```

对位最小值同理。

8.2.1.4 向量 split

- 标量版

```
#define LEN 256  
...  
__nram__ half dst0[LEN];  
__nram__ half dst1[LEN];  
__nram__ half src[2*LEN];  
  
for (int i = 0; i < LEN; ++i) {  
    dst0[i] = src[2 * i];  
    dst1[i] = src[2 * i + 1];  
}
```

Cambricon@155chb

- 向量版

```
#define LEN 256  
...  
__nram__ half dst0[LEN];  
__nram__ half dst1[LEN];  
__nram__ half src[2*LEN];  
__nram__ half mask0[2*LEN] = {1, 0, 1, 0, ..., 1, 0, 1, 0};  
__nram__ half mask1[2*LEN] = {0, 1, 0, 1, ..., 0, 1, 0, 1};  
__bang_collect(dst0, src, mask0, 2 * LEN);  
__bang_collect(dst1, src, mask1, 2 * LEN);
```

8.2.1.5 矩阵乘法

对于矩阵乘法 $C = A * B$, 其中 A 为 $M * N$ 矩阵, B 为 $N * P$ 矩阵, C 为 $M * P$ 矩阵, 用户可以用 `__bang_conv` 来实现。

```
#define M 64
#define N 64
#define P 64
...
__bang_conv(/*output*/C, /*source*/A, /*kernel*/B,
           /*input_channel*/N, /*input_height*/M,
           /*input_width*/1, /*kernel_height*/1,
           /*kernel_width*/1, /*stride_height*/1,
           /*stride_width*/1, /*output_channel*/P,
           /*fix_position*/0);
```

8.2.1.6 置零操作

当需要进行写零操作时, 可以选择 `__bang_write_zero` 接口, 或者 `__nramset` 接口。由于使用的硬件指令不同, `__bang_write_zero` 的性能是要优于 `__nramset` 的, 因此优先选择 `__bang_write_zero` 去实现置零操作。这里可以看出, 可能对于某种计算逻辑, 可以用不同的 Cambricon BANG C 函数去实现, 要根据实际情况如数据大小等, 去选择一种性能最优的方法。

8.2.2 使用融合操作优化性能

Cambricon BANG C 提供了多个融合操作接口, 这类接口不仅可以减少数据中转和 NRAM 空间占用, 还可以利用硬件的指令融合功能达到更高的性能。

- `__bang_fusion` 接口: 可以实现三个输入向量的乘加、乘减、加乘、减乘、减减、加加、加减、减加运算。
- `__bang_fcmpfilter` 接口: 以两个输入向量的比较结果为掩码, 从第三个输入向量中选择掩码值为 1 的数据。比较操作的类型支持: 等于、不等于、小于、小于或等于、大于、大于或等于。
- `__bang fabsmax` 和 `__bang fabsmin` 接口: 分别从输入向量中选出绝对值最大的、最小的数。

8.2.3 使用查表功能实现并行间接访问

Cambricon BANG C 提供了并行查表接口 `__bang_lut_s32` 和 `__bang_lut_s16`，使用这类接口可以实现并行间接访问功能。例如：

```
int table[SIZE];
int offset[SIZE];
int output[SIZE];
...
for (int i = 0; i < SIZE; i++) {
    output[i] = table[offset[i]];
}
```

可以使用 `__bang_lut_s32` 实现并行间接访问：

```
int table[SIZE];
int offset[SIZE];
int output[SIZE];
...
__bang_lut_s32(output, offset, table, SIZE, SIZE);
```

Cambricon@155chb

8.2.4 合理使用核内同步和核间同步接口

Cambricon BANG C 提供的 `_mlvm_sync` 接口可以实现一个 MLU Core 或 Memory Core 内部所有指令流的同步。但是，在实际应用中往往只需要同步某两条指令流，其他流的指令可以继续执行以保证性能。此时，可以选择同步范围更小的同步接口，例如 `__sync_copy_nram_to_dram`、`__sync_copy_sram_to_nram` 和 `__sync_compute` 等接口。

另外，Cambricon BANG C 还提供了 `__sync_cluster`、`__sync_all_ipus`、`__sync_all_mpus` 和 `__sync_all` 用于实现核间同步。在具体使用过程中，用户应当控制同步范围，让尽可能少的 MLU Core 或 Memory Core 参与同步，以保证性能。例如，能用 `__sync_cluster` 的场景就不建议使用 `__sync_all`。

8.2.5 选择合适的接口实现核内指令流并行

在 MLU 硬件中实现同样的功能可以有多种方法，在实际编程时应当根据场景选择合适的接口以实现指令流并行。例如，`__bang_write_value` 在 Compute 流执行，而 `__memset_nram` 在 Move 流执行。如果用户想实现向量运算与 `memset` 的并行执行，应该选择 `__memset_nram`；如果用户想实现 `memset` 与 IO 操作的并行，应该选择 `__bang_write_value`。

8.2.6 利用多核并行计算优化性能

多核并行即利用 MLU 硬件的多芯片、多 Cluster、多核资源，实现三个粒度的并行：数据并行、模型并行、混合并行。

数据并行类似于 SIMD (Single Instruction Multiple Data，单指令多数据)，所有核运行同一份指令，但处理的输入数据不同，从而实现计算并行。模型并行指所有核分工计算同一份输入数据，这需要用户在算法上做等效的计算拆分。混合并行融合了数据并行和模型并行的特点，在 Cluster 之间做数据并行，Cluster 内的多核之间做模型并行。

下面给出使用多核并行进行两个数组相除操作的示例代码。

- BLOCK 版

```
#include<bang.h>
#define LEN 65536

__mlu_entry__ void kernel(half* dst, half* src0, half *src1) {
    __nram__ half src0_nram[LEN];
    __nram__ half src1_nram[LEN];
    __memcpy(src0_nram, src0, LEN * sizeof(half), GDRAM2NRAM);
    __memcpy(src1_nram, src1, LEN * sizeof(half), GDRAM2NRAM);
    for (int i = 0; i < LEN; i++) {
        src1_nram[i] = src0_nram[i] / src1_nram[i];
    }
    __memcpy(dst, src1_nram, LEN * sizeof(half), NRAM2GDRAM);
}
```

- Union8 版

```
#include<bang.h>

#define CORE_NUM 32
#define LEN 65536
#define PER_CORE_LEN (LEN / CORE_NUM)

__mlu_entry__ void kernel(half* dst, half* src0, half *src1) {
    __nram__ half src0_nram[PER_CORE_LEN]; // use macro for constant
    __nram__ half src1_nram[PER_CORE_LEN]; // use macro for constant
    __memcpy(src0_nram, src0 + taskId * PER_CORE_LEN,
             PER_CORE_LEN * sizeof(half), GDRAM2NRAM);
    __memcpy(src1_nram, src1 + taskId * PER_CORE_LEN,
             PER_CORE_LEN * sizeof(half), GDRAM2NRAM);
    for (int i = 0; i < PER_CORE_LEN; i++) {
```

```

    src1_nram[i] = src0_nram[i] / src1_nram[i];
}

__memcpy(dst + taskId * PER_CORE_LEN, src1_nram,
         PER_CORE_LEN * sizeof(half), NRAM2GDRAM);
}

```

8.2.7 增加核间并行度

在使用 Cambricon BANG C 编程过程中应当最大化核间并行度。在如下所示的 UNION1 任务中，MLU Core 1 的 GDRAM2NRAM 方向的 `__memcpy` 需要在 MLU Core 0 的 NRAM2GDRAM 方向的 `__memcpy` 执行完毕以后才能执行，因此原始代码中插入了 `__sync_all` 来实现核间同步。但是 `__sync_all` 强制要求 MLU Core 0 完成全部操作以后，MLU Core 1 才能开始执行。

经过分析可以发现，MLU Core 0 和 MLU Core 1 上都有一些没有依赖关系的操作可以并行执行。经过优化后，MLU Core 1 的一次 `__bang_max` 和 `__memcpy` 被提前到 `__sync_all` 以前，从而将这一部分的处理时间与 MLU Core 0 重叠。

- 优化前

```

if (coreId == 0) {
    __memcpy(nram_data, ldram_data, len * sizeof(half), LDRAM2NRAM);
    __bang_mul_scalar(nram_data, nram_data, 2, len);
    __memcpy(gdram_data, nram_data, len * sizeof(half), NRAM2GDRAM);
}

__sync_all();

if (coreId == 1) {
    __memcpy(nram_data2, ldram_data, len * sizeof(half), LDRAM2NRAM);
    __bang_max(nram_data2, nram_data2, len);
    __memcpy(nram_data3, gdram_data, len * sizeof(half), GDRAM2NRAM);
    __bang_mul_scalar(nram_data2, nram_data3, nram_data2[0], len);
    __memcpy(gdram_data, nram_data2, len * sizeof(half), NRAM2GDRAM);
}

```

- 优化后

```

if (coreId == 0) {
    __memcpy(nram_data, ldram_data, len * sizeof(half), LDRAM2NRAM);
    __bang_mul_scalar(nram_data, nram_data, 2, len);
    __memcpy(gdram_data, nram_data, len * sizeof(half), NRAM2GDRAM);
}

if (coreId == 1) {
    __memcpy(nram_data2, ldram_data, len * sizeof(half), LDRAM2NRAM);
    __bang_max(nram_data2, nram_data2, len);
    __memcpy(nram_data3, gdram_data, len * sizeof(half), GDRAM2NRAM);
    __bang_mul_scalar(nram_data2, nram_data3, nram_data2[0], len);
    __memcpy(gdram_data, nram_data2, len * sizeof(half), NRAM2GDRAM);
}

```

```

__bang_max(nram_data2, nram_data2, len);
}

__sync_all();

if (coreId == 1) {
    __memcpy(nram_data3, gdram_data, len * sizeof(half), GDRAM2NRAM);
    __bang_mul_scalar(nram_data2, nram_data3, nram_data2[0], len);
    __memcpy(gdram_data, nram_data2, len * sizeof(half), NRAM2GDRAM);
}

```

8.2.8 使用 Cluster 间流水

Cluster 间的流水是将一个完整的任务拆解为若干个子任务，每个 Cluster 处理一个子任务，子任务之间可以通过 SRAM 进行数据交换。

以下代码示例用于将 GDRAM 上的输入向量 input 做逐元素的乘加运算，并将计算结果写回到 GDRAM 上的输出向量 output。整个算法先在 Cluster 0 上实现向量乘法，再将中间计算结果拷贝到 Cluster 1 做向量加法运算。

```

#include<bang.h>

#define SIZE 1024

Cambricon@155chb

__nram__ float nram_buf[SIZE];
__mlu_shared__ float sram_buf[4 * SIZE];

__mlu_entry__ void kernel(float* output, float* input) {
    if (clusterId == 0) {
        __memcpy(sram_buf, input, 4 * SIZE * sizeof(float), GDRAM2SRAM);
        __sync_cluster();
        __memcpy(nram_buf, sram_buf + coreId * SIZE, SIZE * sizeof(float), SRAM2NRAM);
        __bang_mul_scalar(nram_buf, nram_buf, 2.0f, SIZE);
        __memcpy(sram_buf + coreId * SIZE, nram_buf, SIZE * sizeof(float), NRAM2SRAM);
        __sync_cluster();
        __memcpy(sram_buf, sram_buf, 4 * SIZE * sizeof(float), SRAM2SRAM, 1);
    }

    __sync_all();

    if (clusterId == 1) {
        __memcpy(nram_buf, sram_buf + coreId * SIZE, SIZE * sizeof(float), SRAM2NRAM);
        __bang_add_scalar(nram_data, nram_data, 2.0f, SIZE);
    }
}

```

```
__memcpy(sram_buf + coreId * SIZE, nram_buf, SIZE * sizeof(float), NRAM2SRAM);
__sync_cluster();
__memcpy(output, sram_buf, 4 * SIZE * sizeof(float), SRAM2GDRAM);
}
}
```

8.3 编译器常用技巧

8.3.1 常数预处理

编译时可以确定的数据写为常量，以减少编译生成的指令条数，提升效率。对于一些可以提前计算的，不随输入数据变化而变化的一些数据，可以提前计算好传入，以减少计算量。

8.3.2 使用位运算

在特定情况下，使用位运算代替部分更耗时的整型标量计算。如当 $N = 2^n$ 时，使用 $i \& (N - 1)$ 代替 $i \% N$ 。

8.3.3 编译 Hint Cambricon@155chb

8.3.3.1 __builtin_assume_aligned

函数原型：

```
void* __builtin_assume_aligned(const void* exp, size_t align)
```

该 Hint 用于告知编译器参数指针是 align 字节对齐，并返回对应指针。使用示例：

```
// 编译器可以假设`char*`res - 8`为 32B 对齐。
void* res = __builtin_assume_aligned(ptr, 32, 8);
```

8.3.3.2 __builtin_assume

函数原型：

```
void __builtin_assume(bool exp)
```

该 Hint 用于告知编译器给定条件为真，如果运行时条件不为真，则产生未定义行为。使用示例：

```
--mlu_func__ int get(int* ptr, int idx) {
    __builtin_assume(idx < 4);
    return ptr[idx];
}
```

8.3.3.3 __builtin_unreachable

函数原型：

```
void __builtin_unreachable(void)
```

该 Hint 用于告知编译器调用位置永远不会被执行，如果运行时执行到 Hint 调用位置，则产生未定义行为。使用示例：

```
switch (size) {
    case 1: return 8;
    case 2: return 16;
    case 4: return 32;
    case 8: return 64;
    default: __builtin_unreachable();
}
```

Cambricon@155chb

8.3.3.4 __restrict__

`__restrict__` 是 C 语言中的一种类型限定符，用于告知编译器一块空间只被该指针指向，不会通过其他指令直接或间接地方式修改该对象的内容。`restrict` 指针在 C 语言中用于缓解指针歧义，使编译器能够进一步进行子表达式删除、指令调度等优化。使用样例：

- 优化前

```
--mlu_entry__ void kernel(float* dst, float* src0, float* src1) {
    dst[0] = src0[0] * src1[0];
    dst[1] = src0[0] * src1[0];
    dst[2] = src0[0] * src1[0] * src0[1];
    dst[3] = src0[0] * src0[1];
    dst[4] = src0[0] * src1[0];
    dst[5] = src1[0];
    return;
}
```

- 优化后

```
--mlu_global__ void kernel(float* __restrict__ dst,
                           float* __restrict__ src0,
                           float* __restrict__ src1) {
    dst[0] = src0[0] * src1[0];
    dst[1] = src0[0] * src1[0];
    dst[2] = src0[0] * src1[0] * src0[1];
    dst[3] = src0[0] * src0[1];
    dst[4] = src0[0] * src1[0];
    dst[5] = src1[0];
    return;
}
```

8.4 访存优化

访存优化的目标是通过优化访存来达到最大化硬件利用率的效果，常用的方法包括减少访存量、计算和 IO 并行、提升带宽利用率。

8.4.1 减少访存数据量

数据在 GDRAM 与片上存储空间之间搬移的代价是非常高的。因此，应当尽可能减小访存数据量。减少访存量的方法主要是片上驻留和计算换访存。

8.4.1.1 片上驻留

MLU Core 读写片上缓存的速度快于读写片外 GDRAM，因此应当将所有高频使用的数据保存在片上，以减少访存时间。

以卷积运算的卷积核处理为例，当卷积核数据量较小时，可以考虑一次将卷积核加载到 WRAM 上，并驻留在 WRAM 上直到计算结束。当卷积核数据量较大时，可以根据 WRAM 存储空间的大小对卷积核进行拆分，并将拆分出的卷积核分到每个 MLU Core 各自的 WRAM 上分别进行计算，再将多个 MLU Core 上的计算结果拼接起来形成最终的计算结果。

如果多核拆分以后 WRAM 空间仍然不够，可以考虑先将一部分卷积核加载到 WRAM 上，而将剩余的卷积核分发到多个 Cluster 的 SRAM 空间，在计算过程中通过卷积核轮转来完成整个计算。[图 8.12 卷积核轮转示意图](#) 以四个 Cluster 的卷积核轮转为例进行说明。图中将卷积核分成四部分，即 Block0、Block1、Block2 和 Block3，分别加载到四个 Cluster 的 SRAM 空间。初始状态下，Block0 被加载到 Cluster0，Block1 被加载到 Cluster1，Block2 被加载到 Cluster2，Block3 被加载到 Cluster3。每个 Cluster 内的四个 MLU Core 从 Cluster 内的 SRAM 中读取卷积核并执行计算任务。在四个 Cluster 的所有 MLU Core 都完成了计算以后，再让四个 Cluster 的 SRAM 之间通过轮转的方式交换卷积核数据。重复上述过程三

次之后，就完成了完整的计算任务。最终，Block3 位于 Cluster0 上，Block0 位于 Cluster1 上，Block1 位于 Cluster2 上，Block2 位于 Cluster3 上。

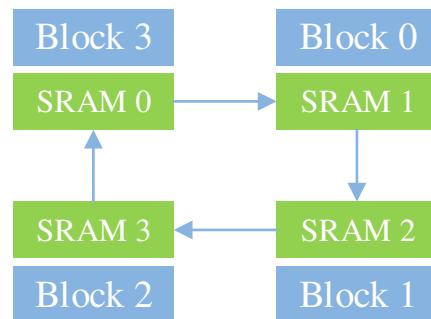
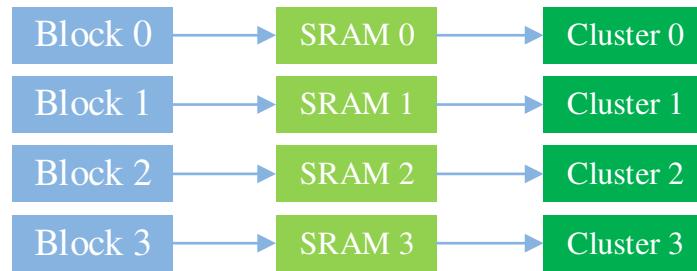


图 8.12: 卷积核轮转示意图

上述的卷积核轮转方法也同样适用于轮转需要存储在 NRAM 上的数据。

8.4.1.2 计算换访存

计算换访存常用于 IO 密集的场景，其方法是通过重新计算某些数据来减少访存数据量。需要注意的是，本方法仅限在访存瓶颈较为严重的情况下使用，否则容易成为负优化。

8.4.2 计算和访存并行

如果无法减少数据的访存量，那么还可以让访存指令与计算指令并行，即用计算时间隐藏访存时间。MLU 硬件的多指令队列特性为计算和访存并行提供了强有力的支持。在具体编程时，应当将有依赖的计算和访存指令在时序上分开，将无依赖的计算和访存指令放在一起并发执行，从而提高硬件利用率和程序性能。

实现计算和访存并行包括以下两种方法：重排代码和软流水。

8.4.2.1 重排代码

重排代码是在不影响程序的语义的前提下重新排布代码执行的顺序。重排代码可以提升计算和访存之间的并行性。下面举一个加法和卷积融合的例子来说明如何重排代码，示例代码中省略了部分不重要参数的定义：

```
__mlu_entry__ void addConvOrigin(half *out_data,
                                  int16_t *in_data,
                                  int16_t *add_data,
                                  int8_t *filter_data) {
    __nram__ half nram_out_data[OUT_DATA_NUM];
    __nram__ int16_t nram_in_data[IN_DATA_NUM];
    __nram__ int16_t nram_add_data[IN_DATA_NUM];
    __wram__ int8_t wram_filter2[FILTER_DATA_NUM];

    __memcpy(wram_filter, filter_data, FILTER_DATA_NUM * sizeof(int8_t), GDRAM2WRAM);
    __memcpy(nram_in_data, in_data, IN_DATA_NUM * sizeof(int16_t), GDRAM2NRAM);
    __memcpy(nram_add_data, add_data, IN_DATA_NUM * sizeof(int16_t), GDRAM2NRAM);

    __bang_add(nram_in_data, nram_in_data, nram_add_data);
    __bang_conv(nram_out_data, nram_in_data, wram_filter, IN_CHANNEL, IN_HEIGHT, IN_WIDTH,
                FILTER_HEIGHT, FILTER_WIDTH, STRIDE_WIDTH, STRIDE_HEIGHT, OUT_CHANNEL, POS);

    __memcpy(out_data, nram_out_data, OUT_DATA_NUM * sizeof(half), NRAM2GDRAM);
}

__mlu_entry__ void addConvOpt(half *out_data,
                             int16_t *in_data,
                             int16_t *add_data,
                             int8_t *filter_data) {
    __nram__ half nram_out_data[OUT_DATA_NUM];
    __nram__ int16_t nram_in_data[IN_DATA_NUM];
    __nram__ int16_t nram_add_data[IN_DATA_NUM];
    __wram__ int8_t wram_filter2[FILTER_DATA_NUM];

    __memcpy(nram_in_data, in_data, IN_DATA_NUM * sizeof(int16_t), GDRAM2NRAM);
    __memcpy(nram_add_data, add_data, IN_DATA_NUM * sizeof(int16_t), GDRAM2NRAM);
    __bang_add(nram_in_data, nram_in_data, nram_add_data);
    __memcpy(wram_filter, filter_data, FILTER_DATA_NUM * sizeof(int8_t), GDRAM2WRAM);
    __sync_io();
    __bang_conv(nram_out_data, nram_in_data, wram_filter, IN_CHANNEL, IN_HEIGHT, IN_WIDTH,
                FILTER_HEIGHT, FILTER_WIDTH, STRIDE_WIDTH, STRIDE_HEIGHT, OUT_CHANNEL, POS);
```

```
__memcpy(out_data, nram_out_data, OUT_DATA_NUM * sizeof(half), NRAM2GDRAM);
}
```

以上原始例子首先将所有的数据加载至片上，然后再进行计算，导致计算和访存之间没有并行。重排后的代码首先将 `__bang_add` 所需的输入数据加载至片上，然后同时执行卷积核的加载和加法的计算。重排代码这种优化手段需要视具体的应用场景而定，目标是让没有依赖的不同流的操作同时执行，提升硬件利用率。

8.4.2.2 软流水

软流水是一种特殊的指令重排，排布的是同一循环内不同循环迭代之间的代码。以下面一个例子说明：

```
#define SIZE 128
#define TOTAL_SIZE (SIZE * 2)

__mlu_entry__ void add0origin(float *a_dram, float *b_dram, float* result_dram) {
    __nram__ float a_nram[SIZE];
    __nram__ float b_nram[SIZE];
    __nram__ float result_nram[SIZE];
    for (int i = 0; i < 10; i++) {
        __memcpy(a_nram, a_dram + i * SIZE, SIZE * sizeof(float), GDRAM2NRAM);
        __memcpy(b_nram, b_dram + i * SIZE, SIZE * sizeof(float), GDRAM2NRAM);
        __bang_add(result_nram, a_nram, b_nram, SIZE);
        __memcpy(result_dram + i * SIZE, result_nram, SIZE * sizeof(float), NRAM2GDRAM);
    }
}

__mlu_entry__ void add0pt(float *a_dram, float *b_dram, float* result_dram) {
    __nram__ float a_nram[TOTAL_SIZE];
    __nram__ float b_nram[TOTAL_SIZE];
    __nram__ float result_nram[TOTAL_SIZE];

    __memcpy(a_nram + 0 * SIZE, a_dram + 0 * SIZE, SIZE * sizeof(float), GDRAM2NRAM);
    __memcpy(b_nram + 0 * SIZE, b_dram + 0 * SIZE, SIZE * sizeof(float), GDRAM2NRAM);
    __bang_add(result_nram + 0 * SIZE, a_nram + 0 * SIZE, b_nram + 0 * SIZE, SIZE);
    __memcpy_async(a_nram + 1 * SIZE, a_dram + 1 * SIZE, SIZE * sizeof(float), GDRAM2NRAM);
    __memcpy_async(b_nram + 1 * SIZE, b_dram + 1 * SIZE, SIZE * sizeof(float), GDRAM2NRAM);

    for (int i = 2; i < 10; i++) {
        __mlvm_sync();
        __memcpy_async(result_dram + (i - 2) * SIZE, result_nram + (i % 2) * SIZE,
```

```
        SIZE * sizeof(float), NRAM2GDRAM);

__bang_add(result_nram + ((i - 1) % 2) * SIZE, a_nram + ((i - 1) % 2) * SIZE,
           b_nram + ((i - 1) % 2) * SIZE, SIZE);

__memcpy_async(a_nram + (i % 2) * SIZE, a_dram + i * SIZE,
               SIZE * sizeof(float), GDRAM2NRAM);

__memcpy_async(b_nram + (i % 2) * SIZE, b_dram + i * SIZE,
               SIZE * sizeof(float), GDRAM2NRAM);

}

__mlvm_sync();

__memcpy_async(result_dram + 8 * SIZE, result_nram + 0 * SIZE,
               SIZE * sizeof(float), NRAM2GDRAM);

__bang_add(result_nram + 1 * SIZE, a_nram + 1 * SIZE, b_nram + 1 * SIZE, SIZE);
__memcpy(result_dram + 9 * SIZE, result_nram + 1 * SIZE,
         SIZE * sizeof(float), NRAM2GDRAM);
}
```

如以上的代码所示，需要将 10 块数据做加法并存出。原始代码中有一个从 0 到 9 的循环，循环迭代变量为 i。循环体中每次完成第 i 块数据加载、计算、存出三个步骤，这三个步骤之间是有数据依赖的，这意味着这三个操作的执行是串行的。而在优化版本的代码中包含一个 2 到 9 的循环，循环迭代变量为 i。循环每次处理的是第 i 块数据的加载、第 i-1 块数据的计算和第 i-2 块数据的存出，这三个步骤之间由于没有数据依赖，可以并行执行，从而提高了硬件的利用率。软流水的本质是将不同循环迭代之间的无关代码放到同一个时间片里执行，使得这些代码之间没有数据依赖，从而尽可能的激发硬件的并行性，达到提高硬件利用率的效果。

将循环从时间上展开，软流水得到的效果如图 8.13 软流水效果图 所示：

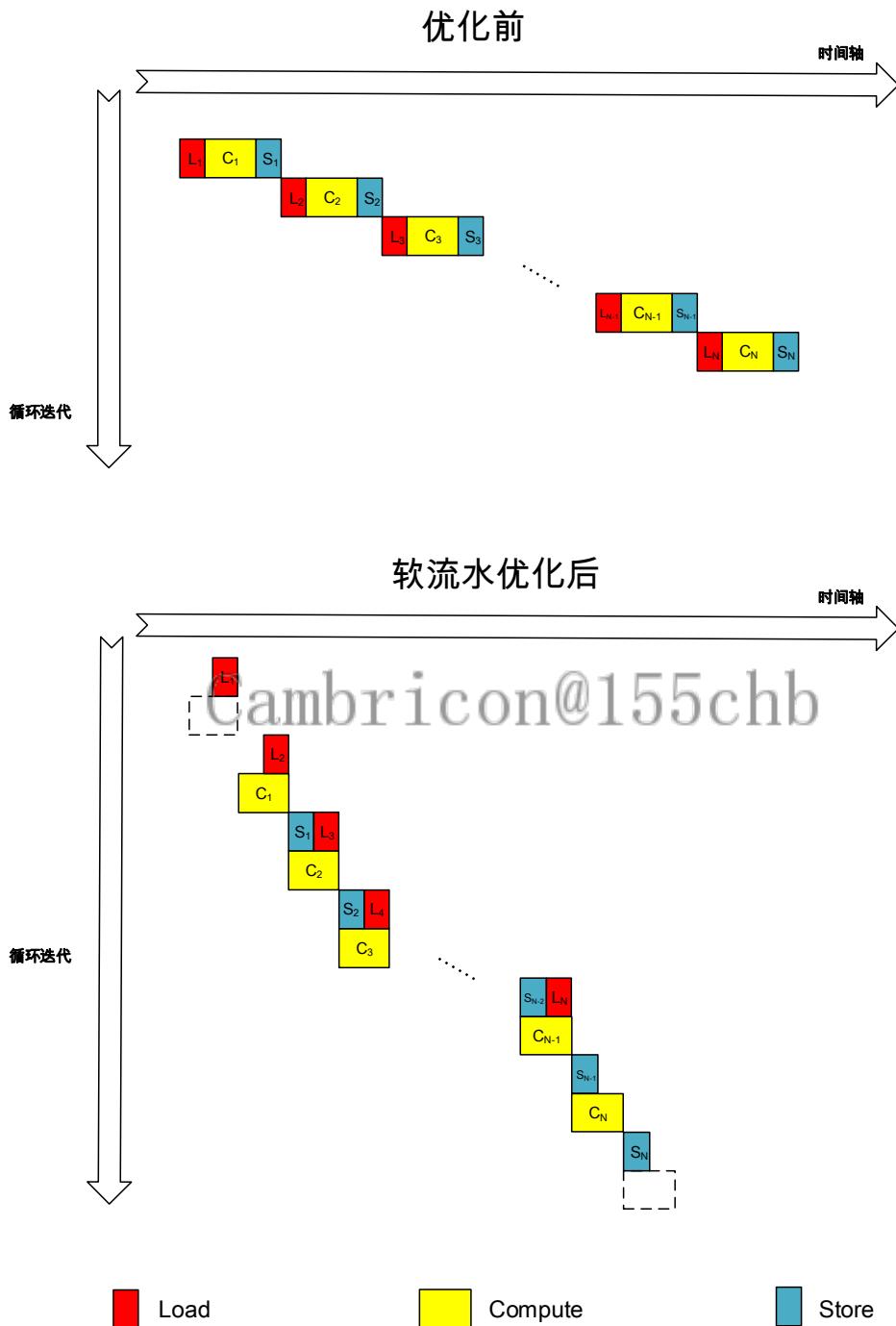


图 8.13: 软流水效果图

对于规模较大的计算任务，可以先将计算任务拆分，再进行软流水。但这种操作可能会降低带宽的利用

率，所以需要权衡软流水带来的收益是否能掩盖拆分计算带来的代价。

8.4.3 提升带宽利用率

针对寒武纪硬件的特点，可以采取对齐访问、张量化访存和阶梯式访存等方式提升带宽利用率。

8.4.3.1 数据对齐

MLU 硬件的访存效率与数据对齐方式有关，且不同代的产品的对齐限制不同。例如在 MLU270 上，地址 128 字节对齐的性能要比 64 字节对齐更好。

8.4.3.2 张量化访存

MLU 硬件对满足对齐且连续存储的数据访存效率非常高。因此，可以采用如下几种方法来提升访存效率：

- 将连续的标量访存换成向量访存可以提高带宽利用率；
- 将多次地址连续的一维向量访存替换成一次更大的一维向量访存；
- 将多次地址不连续，但是地址跳变大小相同且访存大小相同的一维向量访存替换成一次二维向量访存。

下面以一个例子说明如何应用以上方法：

```
#define TOTAL_SIZE (128*10)
__mlu_entry__ void copyOrigin(float *input) {
    __nram__ float a_nram[TOTAL_SIZE];
    __nram__ float b_nram[TOTAL_SIZE];
    __nram__ float c_nram[TOTAL_SIZE];
    for (int i = 0; i < 1280; i++) {
        a_nram[i] = input[i];
    }
    for (int i = 0; i < 10; i++) {
        __memcpy(b_nram + i * 128, input + i * 128, 128 * sizeof(float), GDRAM2NRAM);
    }
    for (int i = 0; i < 10; i++) {
        __memcpy(c_nram + i * 128, input + i * 128, 64 * sizeof(float), GDRAM2NRAM);
    }
}

__mlu_entry__ void copyOpt(float *input) {
    __nram__ float a_nram[TOTAL_SIZE];
    __nram__ float b_nram[TOTAL_SIZE];
    __nram__ float c_nram[TOTAL_SIZE];
    __memcpy(a_nram, input, 1280 * sizeof(float), GDRAM2NRAM);
    __memcpy(b_nram, input, 1280 * sizeof(float), GDRAM2NRAM);
```

```
__memcpy(c_nram, input, 64 * sizeof(float), GDRAM2NRAM, 128, 128, 9);  
}
```

8.4.3.3 阶梯式访存

MLU 具有 GDRAM/LDRAM/SRAM/NRAM/WRAM 多级存储层次，用户可以借助这些存储层次实现递进式、阶梯式的访存。例如：

- 当同一 Cluster 内的多个 MLU Core 使用同样一份数据时，可以先将数据从 GDRAM 拷贝到 SRAM，然后再广播到每个 MLU Core 内的 NRAM/WRAM。
- 当同一 Cluster 内的多个 MLU Core 需要写出数据到 GDRAM 上，且这些数据块地址连续时，可以先将数据从每个 MLU Core 内的 NRAM/WRAM 拷贝到 SRAM，再从 SRAM 拷贝到 DRAM。

8.5 Kernel 级别的性能优化

CNRT 提供了丰富且灵活的队列管理接口、异步 Kernel 执行接口和异步拷贝接口。使用这些接口可以实现硬件资源的高效利用，进而提升 Cambricon BANG C 程序的运行时性能。

Kernel 级别的性能优化主要有三种方式：

- Kernel 融合：指将多个 Kernel 融合成一个 Kernel，避免 Kernel 之间的 IO 操作；
- 多队列并行：指队列级别的 Kernel 并行，属于硬件资源的空间复用；
- 多队列流水：指 Kernel 级别的计算和 IO 隐藏，属于硬件资源的时间复用。

8.5.1 Kernel 融合

在复杂度可控的情况下，可以将多个 Kernel 融合为一个 Kernel。Kernel 融合不仅减少了 Kernel 的启动次数和启动开销，也降低了访存带宽需求和访存延迟，因为 Kernel 之间的数据可以不需要存储到片外再重新加载到片上。

以如下所示的两个 Kernel 为例：

```
#include<bang.h>  
  
#define N 1024  
  
__mlu_entry__ void scale(half* dst, half* src, half scale) {  
    __nram__ half dst_nram[N];  
    __memcpy(dst_nram, src, N * sizeof(half), GDRAM2NRAM);  
    __bang_mul_scalar(dst_nram, dst_nram, scale, N);  
    __memcpy(dst, dst_nram, N * sizeof(half), NRAM2GDRAM);  
}
```

```
--mlu_entry__ void relu(half* dst, half* src) {
    __nram__ half dst_nram[N];
    __memcpy(dst_nram, src, N * sizeof(half), GDRAM2NRAM);
    __bang_active_relu(dst_nram, dst_nram, N);
    __memcpy(dst, dst_nram, N * sizeof(half), NRAM2GDRAM);
}
```

scale 的输出会作为 relu 的输入。将两个 Kernel 融合之后，得到的 Kernel 如下：

```
#include<bang.h>

#define N 1024

__mlu_entry__ void scale_relu(half* dst, half* src, half scale) {
    __nram__ half dst_nram[N];
    __memcpy(dst_nram, src, N * sizeof(half), GDRAM2NRAM);
    __bang_mul_scalar(dst_nram, dst_nram, scale, N);
    __bang_active_relu(dst_nram, dst_nram, N);
    __memcpy(dst, dst_nram, N * sizeof(half), NRAM2GDRAM);
}
```

Cambricon@155chb

8.5.2 多队列并行

多队列并行即多个任务队列执行 Kernel 及其 IO 操作，但每个队列处理不同的数据。

假设下面的例子运行在带有一张 MLU270 (4 个 Cluster) 板卡的机器上，Cambricon BANG C 设备端程序占用 2 个 Cluster，CNRT 创建 2 个任务队列，可以使两批次数据并行计算。

```
cnrtDim3_t dim = {8, 1, 1};
cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_UNION2;

cnrtQueue_t queue_0;
cnrtQueue_t queue_1;
cnrtQueueCreate(&queue_0);
cnrtQueueCreate(&queue_1);

cnrtMemcpyAsync(pDevX0, pHostX0, dataSize, queue_0, cnrtMemcpyHostToDev);
devKernel<<<dim, func_type, queue_0>>>(pDevY0, pDevX0);
cnrtMemcpyAsync(pHostY0, pDevY0, dataSize, queue_0, cnrtMemcpyDevToHost);

cnrtMemcpyAsync(pDevX1, pHostX1, dataSize, queue_1, cnrtMemcpyHostToDev);
devKernel<<<dim, func_type, queue_1>>>(pDevY1, pDevX1);
```

```

cnrtMemcpyAsync(pHostY1, pDevY1, dataSize, queue_1, cnrtMemcpyDevToHost);

cnrtQueueSync(queue_0);
cnrtQueueSync(queue_1);

```

注意：

- 多队列并行既可以并行执行同一个 Kernel，也可以并行执行不同的 Kernel。
- 多队列并行必须要求每个设备端程序都不能占满所有的 MLU 计算资源，否则无法实现并行。

8.5.3 多队列流水

多队列流水利用软流水原理，可以实现计算和 IO 的并行，且每个设备端程序可以占满整个 MLU 计算资源。

下面的例子包含 3 个任务队列，queue_0 仅包含输入数据拷贝任务，queue_1 仅包含设备程序的计算任务，queue_2 仅包含输出数据拷贝任务。整个程序让 3 批次数据实现了流水计算，第 3 阶段实现了计算和 IO 的并行。因而，随着数据批次的增加，可并行的阶段也会增多。为方便理解，将程序写成了表格形式，实际运行代码从上至下逐行按照自左至右的顺序编写即可，如图 8.14 CNRT 多队列流水示意 所示。

cnrtQueue_t queue_0; cnrtQueueCreate(&queue_0); cnrtMemcpyAsync(pDevX0, pHostX0, dataSize, queue_0, cnrtMemcpyHostToDevice); cnrtPlaceNotifier(notifier_01, queue_0);	cnrtQueue_t queue_1; cnrtQueueCreate(&queue_1); cnrtQueueWaitNotifier(notifier_01, queue_1, 0); devKernel<<<dim, func_type, queue_1>>>(pDevY0, pDevX0); cnrtPlaceNotifier(notifier_11, queue_1);	cnrtQueue_t queue_2; cnrtQueueCreate(&queue_2); cnrtQueueWaitNotifier(notifier_11, queue_2, 0); cnrtMemcpyAsync(pHostY0, pDevY0, dataSize, queue_2, cnrtMemcpyDevToHost); cnrtPlaceNotifier(notifier_21, queue_2);
cnrtQueueWaitNotifier(notifier_11, queue_0, 0); cnrtMemcpyAsync(pDevX0, pHostX2, dataSize, queue_0, cnrtMemcpyHostToDevice); cnrtPlaceNotifier(notifier_03, queue_0);	cnrtQueueWaitNotifier(notifier_02, queue_1, 0); devKernel<<<dim, func_type, queue_1>>>(pDevY1, pDevX1); cnrtPlaceNotifier(notifier_12, queue_1);	cnrtQueueWaitNotifier(notifier_11, queue_2, 0); cnrtMemcpyAsync(pHostY0, pDevY0, dataSize, queue_2, cnrtMemcpyDevToHost); cnrtPlaceNotifier(notifier_21, queue_2);
	cnrtQueueWaitNotifier(notifier_03, queue_1, 0); cnrtQueueWaitNotifier(notifier_21, queue_1, 0); devKernel<<<dim, func_type, queue_1>>>(pDevY0, pDevX0); cnrtPlaceNotifier(notifier_13, queue_1);	cnrtQueueWaitNotifier(notifier_12, queue_2, 0); cnrtMemcpyAsync(pHostY1, pDevY1, dataSize, queue_2, cnrtMemcpyDevToHost);
		cnrtQueueWaitNotifier(notifier_13, queue_2, 0); cnrtMemcpyAsync(pHostY2, pDevY0, dataSize, queue_2, cnrtMemcpyDevToHost);

图 8.14: CNRT 多队列流水示意

8.6 使用 Notifier 提高设备利用率

通常在使用 MLU 设备的时候需要获取输入数据，拷贝数据到 MLU 上，在 MLU 上进行计算，将计算结果拷回到主机上，主机进行结果处理这几个步骤。例如：

```
for (i = 0; i < 10; i++) {
    host_input = GetInput();
    cnrtMemcpyAsync(device_input, host_input, dataSize, queue, cnrtMemcpyHostToDev);
    ComputeOnDevice();
    cnrtMemcpyAsync(host_output, device_output, dataSize, queue, cnrtMemcpyDevToHost);
    cnrtQueueSync(queue);
    HostDealWithOutput(host_output);
}
```

这种模式编程简单容易理解，但是性能不是最优的。因为在数据拷贝完成之后主机侧处理的阶段，设备侧实际上是空闲的，没有任务在执行。同时任务下发到设备侧上再次执行也是有延时的。并且 QueueSync 操作会打断设备侧上的 Kernel 执行流水，无法充分利用设备侧的计算资源。如下图上半部分。

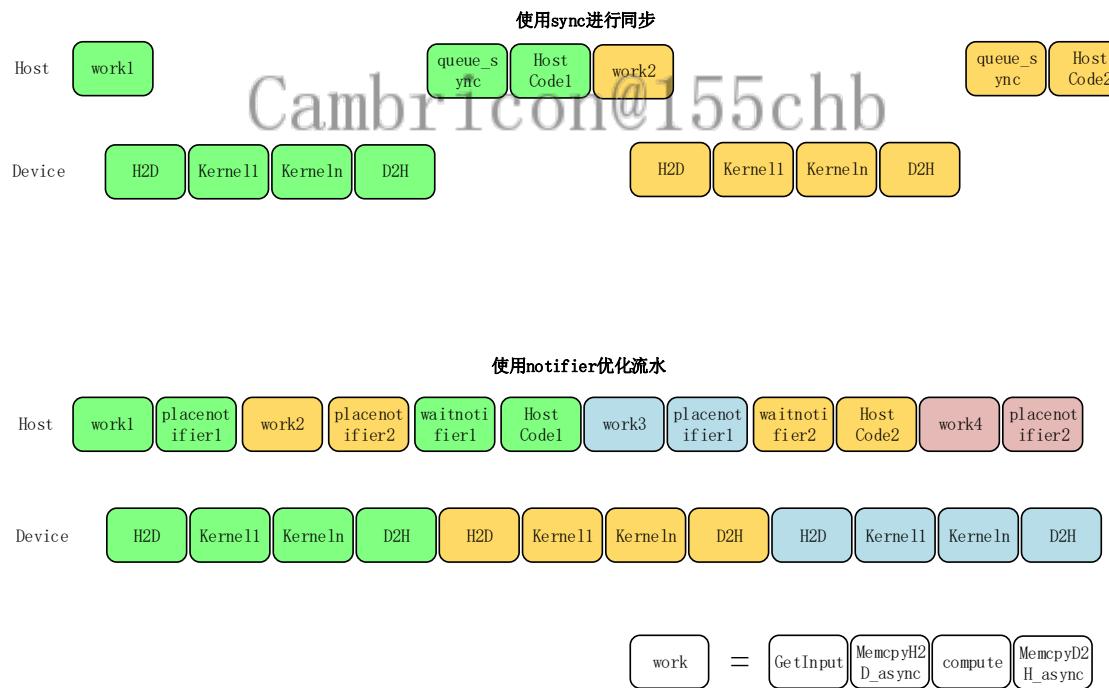


图 8.15: Notifier 流水示意

如果前后两次运算的数据没有依赖关系，可以使用 Notifier 进行优化。通过使用两个 Notifier 和两块缓存交替下发任务来掩盖主机侧开销。

```
host_input1 = GetInput();
cnrtMemcpyAsync(device_input1, host_input1, dataSize, queue, cnrtMemcpyHostToDev);
ComputeOnDevice();
cnrtMemcpyAsync(host_output1, device_output1, dataSize, queue, cnrtMemcpyDevToHost);
cnrtPlaceNotifier(notifier1, queue)

host_input2 = GetInput();
cnrtMemcpyAsync(device_input2, host_input2, dataSize, queue, cnrtMemcpyHostToDev);
ComputeOnDevice();
cnrtMemcpyAsync(host_output2, device_output2, dataSize, queue, cnrtMemcpyDevToHost);
cnrtPlaceNotifier(notifier2, queue)

for (i = 0; i < 8; i+=2) {
    cnrtWaitNotifier(notifier1);
    HostDealWithOutput(host_output1);
    //put new pipe1
    host_input1 = GetInput();
    cnrtMemcpyAsync(device_input1, host_input1, dataSize, queue, cnrtMemcpyHostToDev);
    ComputeOnDevice();
    cnrtMemcpyAsync(host_output1, device_output1, dataSize, queue, cnrtMemcpyDevToHost);
    cnrtPlaceNotifier(notifier1, queue)
    cnrtWaitNotifier(notifier2);
    HostDealWithOutput(host_output2);
    //put new pipe2
    host_input2 = GetInput();
    cnrtMemcpyAsync(device_input2, host_input2, dataSize, queue, cnrtMemcpyHostToDev);
    ComputeOnDevice();
    cnrtMemcpyAsync(host_output2, device_output2, dataSize, queue, cnrtMemcpyDevToHost);
    cnrtPlaceNotifier(notifier2, queue)
}

cnrtWaitNotifier(notifier1);
HostDealWithOutput(host_output1);
cnrtWaitNotifier(notifier2);
HostDealWithOutput(host_output2);
```

Notifier 优化的主要思想是通过 Ping-Pong 模式提高设备利用率，抵消主机侧结果处理的开销和任务再次下发的延时，并且设备侧上 Kernel 流水不会中断。

8.7 向量加法性能调优案例

本节以向量加法为例，介绍基于 Cambricon BANG C 的性能调优方法。整个调优过程分为四个步骤：

第一步：完成一个基于标量加法的实现作为基准程序；

第二步：对基准程序进行向量化，从 SISD (Single Instruction Single Data, 单指令单数据) 到 SIMD (Single Instruction Multiple Data, 单指令多数据)；

第三步：使用软件流水技术，将单核内部的访存和计算并行化处理；

第四步：充分利用 MLU 多核资源，实现多核并行处理。

注意：

在调优过程中，请重点关注 policyFunc (设置任务规模和任务类型等) 和 MLU 设备端代码逻辑。

示例代码的编译命令如下：

```
cncc add.mlu --bang-mlu-arch=mtp_372 -o add -O3
```

注意：

- 本节实验均在 MLU370_X4 板卡上进行，该板卡包含 8 个 Cluster，每个 Cluster 有 4 个 MLU Core；
- 本节实验所用 CN Toolkit 版本为 cn toolkit-v2.4.0，MLU 驱动版本为 v4.15.6。

以下是每个步骤的性能测试结果。可以看到出，MLU 硬件执行时间 (MLU Hardware Time)、MLU 访存效率 (MLU IO Efficiency) 和 MLU 计算效率 (MLU Compute Efficiency) 都随着优化的深入得到了显著地改善。各项指标的计算公式见 第 8.7.5 章 代码附录。

表 8.1: 调优结果

优化技术	MLU 硬件执行时间 (us)	MLU 访存效率	MLU 计算效率
标量	8079206	0.000048	0.000001
向量	4050	0.096260	0.002407
流水	3850	0.101461	0.002537
多核	447	0.873881	0.021847

8.7.1 标量版本

基准程序实现了 float 类型的两组输入数据的加法，每个向量的长度是 10,000,000。

首先给出 policyFunc 的代码，这里设置成了单核任务：

```
void policyFunction(cnrtDim3_t *dim, cnrtFunctionType_t *func_type) {
    *func_type = CNRT_FUNC_TYPE_BLOCK;
    dim->x = 1;
    dim->y = 1;
    dim->z = 1;
    return;
}
```

标量计算核心代码如下：

```
__mlu_func__ void optStep(float *output, float *a, float *b, float *origin_ram, int data_num) {
    if (data_num == 0) {
        return;
    }
    for (int i = 0; i < data_num; i++) {
        *(output + i) = *(a + i) + *(b + i);
    }
    return;
}
```

data_num 为每个核分到的数据量，计算过程为简单的循环遍历相加，计算得到的结果如下：

```
[MLU OPT0_SCALAR      ]: 1
[MLU OPT1_VECTOR       ]: 0
[MLU OPT2_PIPELINE     ]: 0
[MLU OPT3_POLICY       ]: 0
[MLU Hardware Time    ]: 8010691.000 us
[MLU IO Efficiency     ]: 0.000049
[MLU Compute Efficiency]: 0.000001
[MLU Diff Rate         ]: 0.000000
```

可见，执行时间是比较长的，MLU 访存效率和 MLU 计算效率也都很低，后面有很大的优化空间。

8.7.2 向量化版本

接下来，将标量运算转换成向量计算，核心代码如下：

```

__mlu_func__ void optStep(float *output, float *a, float *b, float *origin_ram, int data_num) {
    if (data_num == 0) {
        return;
    }

    uint32_t align_num = NFU_ALIGN_SIZE / sizeof(float);
    uint32_t data_ram_num = MAX_NRAM_SIZE / sizeof(float) / 2 / align_num * align_num;
    float *a_ram = (float *)origin_ram;
    float *b_ram = (float *)a_ram + data_ram_num;

    uint32_t loop_time = data_num / data_ram_num;
    uint32_t rem_ram_num = data_num % data_ram_num;

    for (int i = 0; i < loop_time; i++) {
        // load
        __memcpy(a_ram, a + i * data_ram_num, data_ram_num * sizeof(float), GDRAM2NRAM);
        __memcpy(b_ram, b + i * data_ram_num, data_ram_num * sizeof(float), GDRAM2NRAM);
        // compute
        __bang_add(a_ram, a_ram, b_ram, data_ram_num);
        // store
        __memcpy(output + i * data_ram_num, a_ram, data_ram_num * sizeof(float), NRAM2GDRAM);
    }

    if (rem_ram_num != 0) {
        uint32_t rem_align_num = (rem_ram_num + align_num - 1) / align_num * align_num;
        // load
        __memcpy(a_ram, a + loop_time * data_ram_num, rem_ram_num * sizeof(float), GDRAM2NRAM);
        __memcpy(b_ram, b + loop_time * data_ram_num, rem_ram_num * sizeof(float), GDRAM2NRAM);
        // compute
        __bang_add(a_ram, a_ram, b_ram, rem_align_num);
        // store
        __memcpy(output + loop_time * data_ram_num, a_ram,
                 rem_ram_num * sizeof(float), NRAM2GDRAM);
    }
    return;
}

```

主要思想就是将单个元素的相加，变成了向量式的相加，拷贝和计算都是一条指令处理一个向量，并行性明显提高。受到片上空间容量限制，代码中使用循环做了数据切块处理，一次尽可能处理更多的元素。

计算得到的结果如下：

```
[MLU OPT0_SCALAR      ]: 0
[MLU OPT1_VECTOR       ]: 1
[MLU OPT2_PIPELINE     ]: 0
[MLU OPT3_POLICY       ]: 0
[MLU Hardware Time    ]: 4037.000 us
[MLU IO Efficiency     ]: 0.096761
[MLU Compute Efficiency]: 0.002419
[MLU Diff Rate         ]: 0.000000
```

可见，执行时间、IO 效率和计算效率都有明显地改善。接下来分析一下标量和向量的实现方式。

首先，向量版本可以极大地减少浮点计算指令的数量。向量版本需要的向量运算指令数量为 `loop_time + (rem_ram_num > 0 ? 1 : 0)`，而标量版本所需要的浮点计算指令数量为 `data_num`。

注解：

`--bang_add` 在数据量满足 128 字节（对应代码中的 `NFU_ALIGN_SIZE`）对齐时性能最优，因此，本例中将元素个数对齐到 32。

其次，向量版本可以充分利用访存带宽。标量基准程序对 GDRAM 进行了隐式的数据读写，而且每次只读写一个数据，这种方式效率是很低的。而向量版本程序可以一次把大量数据拷贝到 NRAM 上，极大地提高了带宽利用率。

综上，可以得出以下三个关于 Cambricon BANG C 的优化建议：

小技巧：

- Tip1: 尽量使用 Cambricon BANG C 的向量计算指令。
- Tip2: 尽量使用向量访存指令加载连续的大块数据，避免带宽资源浪费。
- Tip3: 一次性处理的数据最好满足对齐要求，避免计算资源的浪费。

8.7.3 软件流水版本

第 8.4.2 章 计算和访存并行 介绍了软件流水的基本概念和原理，本节介绍如何将软件流水应用到向量加法中。如图 8.16 软件流水示意图 所示：

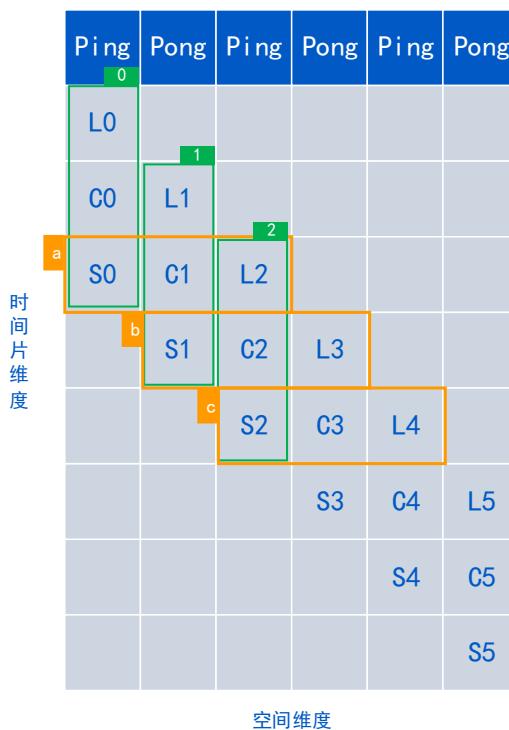


图 8.16: 软件流水示意图

注解:

Cambricon@155chb

- L 代表 Load，加载数据；
- C 代表 Compute，计算数据；
- S 代表 Store，存储数据。

简单来说，软件流水就是对循环中的操作进行调整，使尽可能多的操作可以并行执行。在向量加法案例中，整体逻辑可以分为三部分：Load，Compute 和 Store。为了避免 Load 和 Store 产生冲突，需要引入两块独立的 Ping-Pong 缓冲区。由于 NRAM 大小有限，要进行多次普通的向量计算过程如图中绿色方框所示，顺序执行 0->1->2，即 [L0-C0-S0]->[L1-C1-S1]->[L2-C2-S2]。如果通过软流水技术重新排列，则变成橘红色方框格式，顺序执行 a->b->c，即 [S0-C1-L2]->[S1-C2-L3]->[S2-C3-L4]，可以写成新的循环形式。

注解:

此处说明解释了中间 SCL 循环部分，不考虑导入段 ([L0]->[C0-L1]) 和排空段 ([S4-C5]->[S5]) 的非循环部分。

实现模板如下所示：

```
for (i = 0; i < n + 2; i++) {
    if (i >= 2) {
        S(i - 2);
```

```

    }

    if (i >= 1 && i < n + 1) {
        C(i - 1);
    }

    if (i < n) {
        L(i);
    }

    __sync_all();
}

```

在具体的代码实现中还可以增加一些额外的处理逻辑，比如，将余数段融合到循环段之中。

注解：

- 每个 Load、Compute 和 Store 阶段都可以通过增加 if-else 实现余数段的处理；
- 片上空间划分为 Ping 和 Pong 两块，每块都可以用来临时存放两个输入向量和输出向量。

这里列出了核心代码，完整实现见 第 8.7.5 节 代码附录：

```

__mlu_func__ void optStep(float *output, float *a, float *b, float *origin_ram, int data_num) {
    if (data_num == 0) {
        return;
    }

    // ping: a(out), b // pong: a(out), b
    uint32_t align_num = NFU_ALIGN_SIZE / sizeof(float);
    uint32_t data_ram_num = MAX_NRAM_SIZE / sizeof(float) / 4 / align_num * align_num;
    float *a_ram = origin_ram;
    float *b_ram = a_ram + data_ram_num;

    uint32_t loop_time = data_num / data_ram_num;
    uint32_t rem_ram_num = data_num % data_ram_num;
    int rem_num = 0;
    uint32_t rem_align_num = (rem_ram_num + align_num - 1) / align_num * align_num;
    if (rem_ram_num != 0) {
        rem_num = 1;
    }

    for (int i = 0; i < loop_time + 2 + rem_num; i++) {
        if (i >= 2) {
            // S(i - 2)
            if (i < loop_time + 2 + rem_num - 1 || rem_num == 0) {
                S(output, a_ram, data_ram_num, i - 2);
            } else if (rem_num == 1) {
                S_rem(output, a_ram, data_ram_num, rem_ram_num, loop_time, i - 2);
            }
        }
    }
}

```

```

    }
}

if (i >= 1 && i < loop_time + 1 + rem_num) {
    // C(i - 1)
    if (i < loop_time + 1 + rem_num - 1 || rem_num == 0) {
        C(a_ram, b_ram, data_ram_num, i - 1);
    } else if (rem_num == 1) {
        C_rem(a_ram, b_ram, data_ram_num, rem_align_num, i - 1);
    }
}

if (i < loop_time + rem_num) {
    // L(i)
    if (i < loop_time + rem_num - 1 || rem_num == 0) {
        L(a_ram, a, b_ram, b, data_ram_num, i);
    } else if (rem_num == 1) {
        L_rem(a_ram, a, b_ram, b, data_ram_num, rem_ram_num, loop_time, i);
    }
}

__sync_all_ipu();
}

return;
}

```

Cambricon@155chb

性能测试结果如下：

```

[MLU OPT0_SCALAR      ]: 0
[MLU OPT1_VECTOR       ]: 0
[MLU OPT2_PIPELINE     ]: 1
[MLU OPT3_POLICY       ]: 0
[MLU Hardware Time     ]: 3800.000 us
[MLU IO Efficiency     ]: 0.102796
[MLU Compute Efficiency]: 0.002570
[MLU Diff Rate         ]: 0.000000

```

可以看出，执行时间有了明显改善。不过，由于向量加法的瓶颈在 IO，所以流水起来后，执行时间并没有大幅度的提升。

至此，得到 MLU 性能优化的第四个建议：

小技巧：

- Tip4: 单核心内的访存和计算尽可能做到并行，来隐藏一部分时间。

8.7.4 多核版本

经过上述优化，可以基本保证对大部分算子单核内部的性能没有问题。接下来，还可以利用 MLU 的多核计算资源进一步提升性能。这里主要考虑两个方面：如何将计算任务拆分到多个计算核心上，以及拆分时如何保证性能。

8.7.4.1 多核拆分方法

首先，Cambricon BANG 异构计算平台目前仅支持以下任务类型：

```
#define CNRT_FUNC_TYPE_BLOCK ((cnrtFunctionType_t)1)
#define CNRT_FUNC_TYPE_UNION1 ((cnrtFunctionType_t)4)
#define CNRT_FUNC_TYPE_UNION2 ((cnrtFunctionType_t)8)
#define CNRT_FUNC_TYPE_UNION4 ((cnrtFunctionType_t)16)
#define CNRT_FUNC_TYPE_UNION8 ((cnrtFunctionType_t)32)
#define CNRT_FUNC_TYPE_UNION16 ((cnrtFunctionType_t)64)
```

对于 X4 板卡，任务类型最好设置为 CNRT_FUNC_TYPE_UNION8 或者 CNRT_FUNC_TYPE_UNION1（可以同时启用 8 个），两者均可以充分利用所有的硬件资源。

确定任务类型以后，可以设置任务规模。以 CNRT_FUNC_TYPE_UNION1 为例，设置成 dim->x 为每个 Cluster 包含的 MLU Core 的个数，dim->y 为 Cluster 个数，dim->z 设置为 1，即可将所有 MLU Core 利用起来。具体策略函数实现如下：

```
void policyFunction(cnrtDim3_t *dim, cnrtFunctionType_t *func_type) {
    *func_type = CNRT_FUNC_TYPE_UNION1;
    dim->x = 4;
    dim->y = 8;
    dim->z = 1;
    return;
}
```

上述的拆分方法在 X4 板卡上的执行结果如下：

```
[MLU OPT0_SCALAR      ]: 0
[MLU OPT1_VECTOR       ]: 0
[MLU OPT2_PIPELINE     ]: 1
[MLU OPT3_POLICY       ]: 1
[MLU Hardware Time     ]: 445.000 us
[MLU IO Efficiency     ]: 0.877809
[MLU Compute Efficiency]: 0.021945
[MLU Diff Rate          ]: 0.000000
```

可以看出，硬件执行时间明显减少，访存效率也达到一个比较高的水平。

8.7.4.2 如何保证高性能

对于向量加法这种简单的二元操作算法，可以把两个输入向量和一个输出向量均匀拆分到各个 MLU Core 即可。对于更复杂的算子，可能需要多个高维的输入、输出数据。例如，卷积的数据布局是 $NH_oW_oC_o$ ，这时更倾向于优先拆分高维，即 $N \rightarrow H_o \rightarrow W_o \rightarrow C_o$ ，因为这样能更好的保证数据的连续性，从而保证较高的访存效率。

综上，得到 MLU 性能优化的第五个建议：

小技巧：

- Tip5: 尽量通过调整任务类型和拆分策略将 MLU 所有可利用的资源都利用起来。

另外，由于 MLU 硬件的每条指令流水线的指令队列深度有限，连续发射多条相同类型的指令可能会导致指令队列反压，后续的指令无法继续发射，从而影响并行性。

由此，可以得出第六个优化建议：

小技巧：

- Tip6: 尽量不要连续发射相同类型的向量计算或者访存指令，避免因为指令队列反压导致指令流无法并行。

Cambricon@155chb

8.7.5 代码附录

```
#include <bang.h>

#define ELEM_NUM 10 * 1000 * 1000
#define MAX_NRAM_SIZE 655360
#define NFU_ALIGN_SIZE 128

// need to set one of OPTM0, OPTM1, OPTM2 to 1.
#define OPT0_SCALAR 0
#define OPT1_VECTOR 0
#define OPT2_PIPELINE 1
#define OPT3_POLICY 1

__nram__ char ram[MAX_NRAM_SIZE];

float src1_cpu[ELEM_NUM];
float src2_cpu[ELEM_NUM];
float dst_cpu[ELEM_NUM];
```

```
--mlu_func__ void L(float *a_ram, float *a, float *b_ram,
                     float *b, int data_ram_num, int i) {
    int offset = i % 2 * data_ram_num * 2;
    __memcpy_async(a_ram + offset, a + i * data_ram_num,
                   data_ram_num * sizeof(float), GDRAM2NRAM);
    __memcpy_async(b_ram + offset, b + i * data_ram_num,
                   data_ram_num * sizeof(float), GDRAM2NRAM);
}

--mlu_func__ void C(float *a_ram, float *b_ram, int data_ram_num, int i) {
    int offset = i % 2 * data_ram_num * 2;
    __bang_add(a_ram + offset, a_ram + offset, b_ram + offset, data_ram_num);
}

--mlu_func__ void S(float *output, float *a_ram, int data_ram_num, int i) {
    int offset = i % 2 * data_ram_num * 2;
    __memcpy_async(output + i * data_ram_num, a_ram + offset,
                   data_ram_num * sizeof(float), NRAM2GDRAM);
}

--mlu_func__ void L_rem(float *a_ram, float *a, float *b_ram, float *b,
                      int data_ram_num, int rem_ram_num, int loop_time, int i) {
    int offset = i % 2 * data_ram_num * 2;
    __memcpy_async(a_ram + offset, a + loop_time * data_ram_num,
                   rem_ram_num * sizeof(float), GDRAM2NRAM);
    __memcpy_async(b_ram + offset, b + loop_time * data_ram_num,
                   rem_ram_num * sizeof(float), GDRAM2NRAM);
}

--mlu_func__ void C_rem(float *a_ram, float *b_ram,
                      int data_ram_num, int rem_align_num, int i) {
    int offset = i % 2 * data_ram_num * 2;
    __bang_add(a_ram + offset, a_ram + offset, b_ram + offset, rem_align_num);
}

--mlu_func__ void S_rem(float *output, float *a_ram, int data_ram_num,
                      int rem_ram_num, int loop_time, int i) {
    int offset = i % 2 * data_ram_num * 2;
    __memcpy_async(output + loop_time * data_ram_num, a_ram + offset,
                   rem_ram_num * sizeof(float), NRAM2GDRAM);
}
```

```
--mlu_func__ void optStep(float *output, float *a, float *b,
                           float *origin_ram, int data_num) {
    if (data_num == 0) {
        return;
    }
    #if OPT0_SCALAR
    for (int i = 0; i < data_num; i++) {
        *(output + i) = *(a + i) + *(b + i);
    }
    #elif OPT1_VECTOR
    uint32_t align_num = NFU_ALIGN_SIZE / sizeof(float);
    uint32_t data_ram_num = MAX_NRAM_SIZE / sizeof(float) / 2 / align_num * align_num;
    float *a_ram = (float *)origin_ram;
    float *b_ram = (float *)a_ram + data_ram_num;

    uint32_t loop_time = data_num / data_ram_num;
    uint32_t rem_ram_num = data_num % data_ram_num;

    for (int i = 0; i < loop_time; i++) {
        // load
        __memcpy(a_ram, a + i * data_ram_num, data_ram_num * sizeof(float), GDRAM2NRAM);
        __memcpy(b_ram, b + i * data_ram_num, data_ram_num * sizeof(float), GDRAM2NRAM);
        // compute
        __bang_add(a_ram, a_ram, b_ram, data_ram_num);
        // store
        __memcpy(output + i * data_ram_num, a_ram, data_ram_num * sizeof(float), NRAM2GDRAM);
    }
    if (rem_ram_num != 0) {
        uint32_t rem_align_num = (rem_ram_num + align_num - 1) / align_num * align_num;
        // load
        __memcpy(a_ram, a + loop_time * data_ram_num, rem_ram_num * sizeof(float), GDRAM2NRAM);
        __memcpy(b_ram, b + loop_time * data_ram_num, rem_ram_num * sizeof(float), GDRAM2NRAM);
        // compute
        __bang_add(a_ram, a_ram, b_ram, rem_align_num);
        // store
        __memcpy(output + loop_time * data_ram_num, a_ram,
                 rem_ram_num * sizeof(float), NRAM2GDRAM);
    }
    #elif OPT2_PIPELINE
    // ping: a(out), b || pong: a(out), b
    uint32_t align_num = NFU_ALIGN_SIZE / sizeof(float);
```

```
uint32_t data_ram_num = MAX_NRAM_SIZE / sizeof(float) / 4 / align_num * align_num;  
float *a_ram = origin_ram;  
float *b_ram = a_ram + data_ram_num;  
  
uint32_t loop_time = data_num / data_ram_num;  
uint32_t rem_ram_num = data_num % data_ram_num;  
int rem_num = 0;  
uint32_t rem_align_num = (rem_ram_num + align_num - 1) / align_num * align_num;  
if (rem_ram_num != 0) {  
    rem_num = 1;  
}  
for (int i = 0; i < loop_time + 2 + rem_num; i++) {  
    if (i >= 2) {  
        // S(i - 2)  
        if (i < loop_time + 2 + rem_num - 1 || rem_num == 0) {  
            S(output, a_ram, data_ram_num, i - 2);  
        } else if (rem_num == 1) {  
            S_rem(output, a_ram, data_ram_num, rem_ram_num, loop_time, i - 2);  
        }  
    }  
    if (i >= 1 && i < loop_time + 1 + rem_num) {  
        // C(i - 1)  
        if (i < loop_time + 1 + rem_num - 1 || rem_num == 0) {  
            C(a_ram, b_ram, data_ram_num, i - 1);  
        } else if (rem_num == 1) {  
            C_rem(a_ram, b_ram, data_ram_num, rem_align_num, i - 1);  
        }  
    }  
    if (i < loop_time + rem_num) {  
        // L(i)  
        if (i < loop_time + rem_num - 1 || rem_num == 0) {  
            L(a_ram, a, b_ram, b, data_ram_num, i);  
        } else if (rem_num == 1) {  
            L_rem(a_ram, a, b_ram, b, data_ram_num, rem_ram_num, loop_time, i);  
        }  
    }  
    __sync_all_ipu();  
}  
#endif  
return;  
}
```

```
--mlu_func__ void add(float *output,
                      const float *a,
                      const float *b,
                      const int data_num) {
    if (coreId == 0x80) {
        return;
    }

    uint32_t task_dim = taskDim;
    uint32_t task_id = taskId;
    uint32_t data_per_core = data_num / task_dim;
    uint32_t data_last_core = data_per_core + data_num % task_dim;

    float *a_fix = (float *)a + task_id * data_per_core;
    float *b_fix = (float *)b + task_id * data_per_core;
    float *output_fix = (float *)output + task_id * data_per_core;

    if (task_id != task_dim - 1) {
        optStep(output_fix, a_fix, b_fix, (float *)ram, data_per_core);
    } else {
        optStep(output_fix, a_fix, b_fix, (float *)ram, data_last_core);
    }
}

--mlu_entry__ void kernel(float* dst, float* src0, float* src1) {
    add((float *)dst, (float *)src0, (float *)src1, ELEM_NUM);
    return;
}

// policy function
void policyFunction(cnrtDim3_t *dim, cnrtFunctionType_t *func_type) {
    #if OPT3_POLICY
    *func_type = CNRT_FUNC_TYPE_UNION1;
    dim->x = 4;
    dim->y = 8;
    dim->z = 1;
    #else
    *func_type = CNRT_FUNC_TYPE_BLOCK;
    dim->x = 1;
    dim->y = 1;
    dim->z = 1;
    #endif
}
```

```
return;
}

int main() {
    CNRT_CHECK(cnrtSetDevice(0));
    cnrtNotifier_t st, et;
    CNRT_CHECK(cnrtNotifierCreate(&st));
    CNRT_CHECK(cnrtNotifierCreate(&et));
    cnrtQueue_t queue;
    CNRT_CHECK(cnrtQueueCreate(&queue));

    cnrtDim3_t dim;
    cnrtFunctionType_t func_type;
    policyFunction(&dim, &func_type);

    // 1.0f + 1.0f = 2.0f
    for (unsigned i = 0; i < ELEM_NUM; ++i) {
        src1_cpu[i] = 1.0f;
        src2_cpu[i] = 1.0f;
    }
    float* src1_mlu = NULL;
    float* src2_mlu = NULL;
    float* dst_mlu = NULL;
    CNRT_CHECK(cnrtMalloc((void **)&src1_mlu, ELEM_NUM * sizeof(float)));
    CNRT_CHECK(cnrtMalloc((void **)&src2_mlu, ELEM_NUM * sizeof(float)));
    CNRT_CHECK(cnrtMalloc((void **)&dst_mlu, ELEM_NUM * sizeof(float)));
    CNRT_CHECK(cnrtMemcpy(src1_mlu, src1_cpu, ELEM_NUM * sizeof(float),
                         cnrtMemcpyHostToDev));
    CNRT_CHECK(cnrtMemcpy(src2_mlu, src2_cpu, ELEM_NUM * sizeof(float),
                         cnrtMemcpyHostToDev));
    CNRT_CHECK(cnrtPlaceNotifier(st, queue));
    kernel<<<dim, func_type, queue>>>(dst_mlu, src1_mlu, src2_mlu);
    CNRT_CHECK(cnrtPlaceNotifier(et, queue));
    CNRT_CHECK(cnrtQueueSync(queue));
    CNRT_CHECK(cnrtMemcpy(dst_cpu, dst_mlu, ELEM_NUM * sizeof(float),
                         cnrtMemcpyDevToHost));
    float latency;
    CNRT_CHECK(cnrtNotifierDuration(st, et, &latency));
    CNRT_CHECK(cnrtFree(src1_mlu));
    CNRT_CHECK(cnrtFree(src2_mlu));
    CNRT_CHECK(cnrtFree(dst_mlu));
    CNRT_CHECK(cnrtQueueDestroy(queue));
```

```
float diff = 0.0;
float baseline = 2.0;
for (unsigned i = 0; i < ELEM_NUM; ++i) {
    diff += fabs(dst_cpu[i] - baseline);
}
double theory_io = ELEM_NUM * 4.0 * 3.0; // bytes
double theory_ops = ELEM_NUM * 4.0; // ops
// ops_per_core/ns * core_num_per_cluter * cluster_num
double peak_compute_force = 128 * 4 * 8;
double io_bandwidth = 307.2; // bytes/ns
double io_efficiency = theory_io / (latency * 1000) / io_bandwidth;
double cp_efficiency = theory_ops / (latency * 1000) / peak_compute_force;
printf("[MLU OPT0_SCALAR]: %d \n", OPT0_SCALAR);
printf("[MLU OPT1_VECTOR]: %d \n", OPT1_VECTOR);
printf("[MLU OPT2_PIPELINE]: %d \n", OPT2_PIPELINE);
printf("[MLU OPT3_POLICY]: %d \n", OPT3_POLICY);
printf("[MLU Hardware Time]: %.3f us\n", latency);
printf("[MLU IO Efficiency]: %f\n", io_efficiency);
printf("[MLU Compute Efficiency]: %f\n", cp_efficiency);
printf("[MLU Diff Rate]: %f\n", diff);
return 0;
}
```

Cambricon@155chb



9 附录

9.1 C++ 语言扩展

本节主要介绍 Cambricon BANG 异构并行编程模型对 C/C++ 做的语言扩展。

9.1.1 预处理符号

9.1.1.1 __BANG_ARCH__

编写设备侧代码时，实现同样的功能可能在不同的架构上有不同的接口实现，此时可以使用 `__BANG_ARCH__` 宏进行区分。

9.1.1.2 __MLU_NRAM_SIZE__

`__MLU_NRAM_SIZE__` 表示特定架构的 NRAM 容量，单位是 KB。在编写设备侧代码时，可以使用 `__MLU_NRAM_SIZE__` 声明 NRAM 空间的数组。

9.1.1.3 __MLU_SRAM_SIZE__

`__MLU_SRAM_SIZE__` 表示特定架构的 SRAM 容量，单位是 KB。在编写设备侧代码时，可以使用 `__MLU_SRAM_SIZE__` 声明 SRAM 空间的数组。

9.1.1.4 __MLU_WRAM_SIZE__

`__MLU_WRAM_SIZE__` 表示特定架构的 WRAM 容量，单位是 KB。在编写设备侧代码时，可以使用 `__MLU_WRAM_SIZE__` 声明 WRAM 空间的数组。

9.1.2 函数修饰符

9.1.2.1 __mlu_host__

`__mlu_host__` 修饰的函数表示可以在主机侧调用，通常与 `__mlu_func__` 属性同时使用，表示被修饰的函数可以同时被主机侧和设备侧调用。对于不被本节所述的任何函数修饰符修饰的函数，仅能在主机侧调用。

9.1.2.2 __mlu_builtin__

`__mlu_builtin__` 用于修饰 CNCC 编译器的内建函数，使用该修饰符的函数默认会被内联。

9.1.2.3 __mlu_entry__

`__mlu_entry__` 用于修饰设备侧的 Kernel 入口函数。Kernel 函数只能由主机侧调用，不支持在设备侧调用。

注意：

在调用 Kernel 函数时，必须指定任务类型、任务规模和执行队列。

Kernel 的执行是异步的，也就是说主机侧的函数调用会在 Kernel 函数执行完毕前返回。Kernel 函数没有返回值，输入输出只能通过参数传递。

每个 Kernel 都有唯一的入口函数，入口函数还可以调用其他设备侧的函数，从而实现模块化设计。Kernel 的声明包括内建函数声明和 C/C++ 语言声明。

9.1.2.4 __mlu_func__

`__mlu_func__` 用于修饰设备侧的函数。被 `__mlu_func__` 修饰的函数默认会被内联。

注意：

递归函数会忽略 `__mlu_func__` 修饰符，因为递归函数无法被内联。

9.1.3 地址空间修饰符

Cambricon BANG C 语言提供了地址空间修饰符用来描述设备侧变量所在的地址空间。在设备侧函数内部也可以声明没有任何地址空间修饰符的局部变量，这类变量存放在栈空间。

注意：

设备侧的全局变量必须指定地址空间修饰符，没有地址空间修饰符的全局变量会被当成主机侧变量处理。

9.1.3.1 __nram__

`__nram__` 修饰符表示被修饰的变量位于 NRAM 地址空间，NRAM 地址空间的变量可以声明在函数内部，也可以声明在全局作用域。

注意：

目前所有的向量运算指令的输入、输出数据必须位于 NRAM 地址空间。

由 `__nram__` 修饰的变量的生存期为一个 Task，不支持 Task 之间通过 NRAM 通信。NRAM 空间的变量不会被自动初始化，用户需要显式设置初值。不同 MLU Core 的 NRAM 空间互相隔离。NRAM 空间的变量起始地址默认 64 字节对齐，用户可以通过 `__attribute__((align()))` 属性修改对齐方式。

9.1.3.2 __wram__

`__wram__` 修饰符表示被修饰的变量位于 WRAM 地址空间，WRAM 地址空间的变量可以声明在函数内部，也可以声明在全局作用域。

注意：

目前所有的卷积运算、矩阵-向量乘法运算、积分运算和直方图运算的卷积核数据必须位于 WRAM 地址空间。

由 `__wram__` 修饰的变量的生存期为一个 Task，不支持 Task 之间通过 WRAM 通信。WRAM 空间的变量不支持初始化。不同 MLU Core 的 WRAM 空间互相隔离。WRAM 空间的变量起始地址默认 64 字节对齐，用户可以通过 `__attribute__((align()))` 属性修改对齐方式。

如下给出一段计算卷积的代码示例，在计算 conv 前，需要将输入数据搬移至 NRAM 中，同时 filter 需要搬移至 WRAM 中。

```
--mlu_entry__ void ConvKernel(half* out_data,
                           half* in_data,
                           half* filter_data,
                           int in_channel,
                           int in_height,
                           int in_width,
                           int filter_height,
                           int filter_width,
                           int stride_height,
                           int stride_width,
                           int out_channel) {

    __nram__ half nram_out_data[OUT_DATA_NUM];
    __nram__ half nram_in_data[IN_DATA_NUM];
    __wram__ half wram_filter[FILTER_DATA_NUM];
```

```
__memcpy(nram_in_data, in_data, IN_DATA_NUM * sizeof(half), GDRAM2NRAM);
__memcpy(wram_filter, filter_data, FILTER_DATA_NUM * sizeof(half), GDRAM2WRAM);

__bang_conv(nram_out_data, nram_in_data, wram_filter, IN_CHANNEL,
            IN_HEIGHT, IN_WIDTH, FILTER_HEIGHT, FILTER_WIDTH,
            STRIDE_WIDTH, STRIDE_HEIGHT, OUT_CHANNEL);

__memcpy(out_data, nram_out_data, OUT_DATA_NUM * sizeof(half), NRAM2GDRAM);
}
```

9.1.3.3 __lDRAM__

`__lDRAM__` 修饰符表示被修饰的变量位于 LDRAM 地址空间，LDRAM 地址空间的变量可以声明在函数内部，也可以声明在全局作用域。由 `__lDRAM__` 修饰的变量的生存期是一个 Task。LDRAM 空间的变量不会被自动初始化，用户需要显式设置初值。LDRAM 地址空间为每个 Task 的私有存储空间，不支持 Task 之间通过 LDRAM 通信。LDRAM 空间的变量起始地址默认 64 字节对齐，用户可以通过 `__attribute__((align()))` 属性修改对齐方式。LDRAM 空间的大小可以通过驱动配置。

9.1.3.4 __mlu_const__

`__mlu_const__` 修饰符用于修饰常量数据，对应的常量在 Kernel 启动时存放在 GDRAM 上，被所有的 Task 共享。访问 `__mlu_const__` 修饰的常量可以经过 L2 Cache 加速。

注意：

由 `__mlu_const__` 修饰的常量必须声明在全局作用域，不支持在函数内部使用 `__mlu_const__`。

`__mlu_const__` 修饰的变量不会被自动初始化，用户需要显式设置初值。由 `__mlu_const__` 修饰的变量起始地址默认 64 字节对齐，用户可以通过 `__attribute__((align()))` 属性修改对齐方式。

9.1.3.5 __mlu_device__

`__mlu_device__` 修饰符表示被修饰的变量位于 GDRAM 地址空间。GDRAM 空间的变量可以声明在函数内部，也可以声明在全局作用域。GDRAM 空间的变量不会被自动初始化，用户需要显式设置初值。GDRAM 空间的变量的生存期为整个 Kernel，可以被所有 Task 共享。用户可以借助 GDRAM 变量完成原子操作。GDRAM 空间的变量起始地址默认 64 字节对齐，用户可以通过 `__attribute__((align()))` 属性修改对齐方式。

GDRAM 可以作为主机侧与设备侧的数据通信，二者可以通过 `cnrtMemcpy` 接口完成数据传递。如下示例所示。

```
cnrtMemcpy(mlu_result, d_c, data_num * sizeof(half), cnrtMemcpyDevToHost);  
...  
cnrtMemcpy(d_a, h_a_half, data_num * sizeof(half), cnrtMemcpyHostToDev);
```

9.1.3.6 __mlu_shared__

`__mlu_shared__` 修饰符表示被修饰的变量位于 SRAM 地址空间。SRAM 空间的变量可以声明在函数内部，也可以声明在全局作用域。SRAM 空间的变量不支持初始化。SRAM 空间的变量的生存期为整个 Kernel，可以被所有 Task 共享。SRAM 空间的变量起始地址默认 64 字节对齐，用户可以通过 `__attribute__((align()))` 属性修改对齐方式。

`__mlu_shared__` 关键字的语法规则如下：

1. 在 `__mlu_entry__` 函数体中声明 SRAM 变量。

```
__mlu_entry__ void kernel() {  
    __mlu_shared__ int array1[100]; // OK  
    static __mlu_shared__ int array2[100]; // OK  
    // error: initialization is not supported for __mlu_shared__ variables  
    __mlu_shared__ int array3[100] = {0};  
    // error: initialization is not supported for __mlu_shared__ variables  
    __mlu_shared__ int array4[100] = foo();  
    __mlu_shared__ half var1; // OK  
    // error: initialization is not supported for __mlu_shared__ variables  
    __mlu_shared__ half var2 = 1.0;  
}
```

2. `__mlu_func__` 函数体中声明 SRAM 变量。

```
__mlu_func__ void func() {  
    __mlu_shared__ int array1[100]; // OK  
    static __mlu_shared__ int array2[100]; // OK  
    // error: initialization is not supported for __mlu_shared__ variables  
    __mlu_shared__ int array3[100] = {0};  
    // error: initialization is not supported for __mlu_shared__ variables  
    __mlu_shared__ int array4[100] = foo();  
    __mlu_shared__ half var1; // OK  
    // error: initialization is not supported for __mlu_shared__ variables  
    __mlu_shared__ half var2 = 1.0;  
}
```

3. 在文件作用域中定义全局 SRAM 变量。

```
// error: initialization is not supported for __mlu_shared__ variables
static __mlu_shared__ int array1[100] = {0};

static __mlu_shared__ int array2[100]; // OK

// error: initialization is not supported for __mlu_shared__ variables
static __mlu_shared__ int array3[] = {0};
// error: initialization is not supported for __mlu_shared__ variables
static __mlu_shared__ int array4[100] = foo();
static __mlu_shared__ half var1; // OK
// error: initialization is not supported for __mlu_shared__ variables
static __mlu_shared__ half var2 = 1.0;

__mlu_entry__ void kernel() {
}
```

4. 跨越文件作用域定义全局 SRAM 变量。

```
extern __mlu_shared__ int array1[]; // OK
// error: __shared__ variable 'array2' cannot be 'extern'
extern __mlu_shared__ int array2[100];
// warning: 'extern' variable has an initializer [-Wextern-initializer]
extern __mlu_shared__ int array3[] = {0};
// error: __mlu_shared__ variable 'var1' cannot be 'extern'
extern __mlu_shared__ half var1;
// error: __mlu_shared__ variable 'var2' cannot be 'extern'
extern __mlu_shared__ half var2 = 1.0;

__mlu_entry__ void kernel() {
}
```

9.1.4 数据类型支持

Cambricon BANG C 支持的基础数据类型如下表所示。

表 9.1: Cambricon BANG C 支持的基础数据类型

基本数据类型	长度	描述
int8_t	1 byte	1 字节整数。
uint8_t	1 byte	1 字节无符号整数。
int16_t	2 bytes	2 字节整数。
uint16_t	2 bytes	2 字节无符号整数。
int32_t	4 bytes	4 字节整数。
uint32_t	4 bytes	4 字节无符号整数。
half	2 bytes	半精度浮点数据类型，采用 IEEE-754 fp16 格式。
float	4 bytes	IEEE-754 fp32 格式浮点类型，mtp_100 仅支持 float 类型转换，(m)tp_220/(m)tp_270/(m)tp_290/tp_322/mtp_372 支持 float 所有操作。
char	1 byte	对应 C 语言 char 类型。
bfloat16_t	2 bytes	脑浮点 (brain floating point) 格式，包括 1bit 符号位，8bits 指数位，7bits 尾数位，仅支持 MLUv03 及后续硬件架构。
bool	1 byte	对应 C 语言 bool 类型。

对于浮点数常量，如果数字以“f”结尾，代表数字为 float 类型，否则为 half。例如，0.1f 是 float 类型，0.1 则为 half 类型。当超过数据类型的最大范围时，发生溢出，将会得到数据类型范围的最大值。当一个浮点数小于一个正常的浮点数值时，向下溢出，这个数字可能被更新为 0。

此外，Cambricon BANG C 为了方便用户描述任务规模，还定义了 cnrtDim3_t 复合数据类型：

```
typedef struct {
    unsigned int x;
    unsigned int y;
    unsigned int z;
} cnrtDim3_t;
```

9.1.5 内建变量

Cambricon BANG C 提供了一系列内建变量用于在设备侧编程时获取任务类型、任务规模、硬件规模等信息。

9.1.5.1 clusterDim

`clusterDim` 用于获取参与当前 Kernel 运算的 Cluster 数量。对于 UnionN (N=1, 2, 4, 8, ...) 任务, `clusterDim = N`, 对于 Block 任务, `clusterDim = 0`。

9.1.5.2 clusterId

`clusterId` 用于获取当前任务所在的逻辑 Cluster 编号, 取值范围是 [0, `clusterDim - 1`]。

9.1.5.3 coreDim

`coreDim` 用于获取一个 Cluster 内包含的 MLU Core 数量, 对于 tp_322 架构, `coreDim=1`, 其他架构恒为 4。

9.1.5.4 coreId

Cambricon@155chb

`coreId` 用于获取当前任务所在的 MLU Core 或者 Memory Core 在一个 Cluster 内的逻辑编号。对于 MLU Core, `coreId` 的取值范围是 [0, `coreDim - 1`]; 对于 Memory Core, `coreId` 恒等于 128。

9.1.5.5 taskDimX

`taskDimX` 用于获取当前 Kernel 在 X 维度的大小, 其值与主机侧通过 `cnrtDim3_t` 设置的 X 值相同。

9.1.5.6 taskDimY

`taskDimY` 用于获取当前 Kernel 在 Y 维度的大小, 其值与主机侧通过 `cnrtDim3_t` 设置的 Y 值相同。

9.1.5.7 taskDimZ

`taskDimZ` 用于获取当前 Kernel 在 Z 维度的大小, 其值与主机侧通过 `cnrtDim3_t` 设置的 Z 值相同。

9.1.5.8 taskDim

taskDim 用于获取当前 Kernel 的任务总数，即 $\text{taskDim} = \text{taskDimX} * \text{taskDimY} * \text{taskDimZ}$ 。

9.1.5.9 taskIdX

taskIdX 用于获取当前任务在三维任务网络中的 X 方向的坐标，其取值范围为 $[0, \text{taskDimX} - 1]$ 。

9.1.5.10 taskIdY

taskIdY 用于获取当前任务在三维任务网络中的 Y 方向的坐标，其取值范围为 $[0, \text{taskDimY} - 1]$ 。

9.1.5.11 taskIdZ

taskIdZ 用于获取当前任务在三维任务网格中的 Z 方向的坐标，其取值范围为 $[0, \text{taskDimZ} - 1]$ 。

9.1.5.12 taskId

taskId 用于获取当前任务在三维任务网格中的一维线性坐标，即 $\text{taskId} = \text{taskIdZ} * \text{taskDimY} * \text{taskDimX} + \text{taskIdY} * \text{taskDimX} + \text{taskIdX}$ 。

Cambricon@155chb

9.1.6 多核同步接口

在 Cambricon BANG C 语言层封装了多个同步原语来实现核间同步。

9.1.6.1 __sync_cluster()

`__sync_cluster()` 用来同步一个 Cluster 内的所有 MLU Core 和 Memory Core，只有当所有的 MLU Core 和 Memory Core 都到达同步后，才会继续执行后面的指令。对于 Block 任务，`__sync_cluster()` 无效。

9.1.6.2 __sync_all_ipu()

`__sync_all_ipu()` 用来同步执行一轮迭代任务的所有 MLU Core，只有当所有的 MLU Core 都到达同步点后，才能继续执行后面的指令。参与同步的 MLU Core 数量与任务类型相关，对于 Block 任务，只有一个 MLU Core 参与同步；对于 UnionN ($N=1,2,4,8,\dots$) 任务，有 $\text{coreDim} * N$ 个 MLU Core 参与同步。

9.1.6.3 __sync_all_mpu()

`__sync_all_mpu()` 用来同步 UnionN (N=1,2,4,8,...) 任务中的 N 个 Memory Core，只有当 N 个 Memory Core 都到达同步点以后，Memory Core 上才能继续执行后续的指令。对于 Block 任务或者计算能力 1.x 硬件，`__sync_all_mpu()` 无效。

9.1.6.4 __sync_all()

`__sync_all()` 用来同步参与 Kernel 运算的所有 MLU Core 和 Memory Core，只有当所有的 MLU Core 和 Memory Core 都到达同步点以后，才能继续执行后续的指令。对于 Block 任务，只有一个 MLU Core 参与同步。对于 UnionN (N=1,2,4,8,...) 任务，参与同步的核数与具体架构有关：计算能力 1.x 硬件没有 Memory Core，共计 $N * \text{coreDim}$ 个 MLU Core 参与同步；对于计算能力 2.x 以及后续硬件，会有 $N * \text{coreDim}$ 个 MLU Core 和 N 个 Memory Core 参与同步。

9.1.7 地址空间识别函数

9.1.7.1 __is_nram()

判断输入地址是否位于 NRAM 地址范围。

9.1.7.2 __is_wram()

判断输入地址是否位于 WRAM 地址范围。

9.1.7.3 __is_sram()

判断输入地址是否位于 SRAM 地址范围。

9.1.8 格式化输出

在设备侧代码中可以使用 `__bang_printf` 和 `printf` 实现格式化输出，输出结果默认打印到控制台。其中，`__bang_printf` 在功能上与 `printf` 等价，二者支持的格式化字符支持情况也完全相同。Cambricon BANG C 中的格式化字符规范与 C/C++ 基本相同，在兼容 C/C++ 大部分格式化字符规范的前提下，新增了 Cambricon BANG C 特有的格式化字符支持。

Cambricon BANG C 支持的格式化字符串形式如下：

```
%[标志字段][宽度字段][.精度字段][长度字段] 类型字段
```

格式化字符串以百分号开始，以类型字段结束，除了百分号和类型字段以外，其他字段均为可选字段，注意精度字段前面需要有英文句号。

Cambricon BANG C 支持 表 9.2 标志字段的含义 所示的标志字段。

表 9.2: 标志字段的含义

标志字段	含义
-	左对齐（默认右对齐）。
+	给正数附加符号前缀（默认正数没有符号前缀）。
空格	给正数附加空格前缀（默认正数没有任何前缀）。
0	在指定了宽度字段且为右对齐时，前缀补 0（默认为空格，当使用“-”标志指定了左对齐时，0 标志失效）。
#	使用“备选格式”，对于 g 和 G 类型，不省略小数点部分最后的 0；对于 f、F、e、E、g、G 类型，总是输出小数点；对于 o、x、X 类型，分别在非零数值前面附加 0、0x、0X 前缀。

宽度字段指定输出字符的最小长度，长度不足的输出将使用填充字符补齐，填充字符及对齐方式使用上述的 0 标志和-标志确定，超长的输出不受影响。

精度字段指定输出字符的最大长度，对于浮点类型来说，精度字段指定了小数点后的最长有效位数；对于字符串来说，精度字段指定了输出的最大的字符数。

Cambricon BANG C 支持 表 9.3 长度字段的含义 所示的长度字段。

表 9.3: 长度字段的含义

长度字段	含义
hh	用于将 char 类型参数转换为 int 型输出
h	用于将 short 型参数转换为 int 型输出
ll	用于输出 long 型参数，对于浮点类型无效
l	用于输出 long long 类型参数

Cambricon BANG C 支持 表 9.4 类型字段的含义 所示的类型字段。

表 9.4: 类型字段的含义

类型字段	含义
%hhd	按照 int8_t 类型打印整型参数
%hhu	按照 uint8_t 类型打印整型参数
%hd	按照 int16_t 类型打印整型参数
%hu	按照 uint16_t 类型打印整型参数
%d	按照 int32_t 类型打印整型参数
%u	按照 uint32_t 类型打印整型参数
%hf	按照 half 类型打印浮点参数
%f	按照 float 类型打印浮点参数
%c	按照 unsigned char 类型打印整型参数
%s	按照字符串形式打印参数
%p	按照十六进制形式打印指针
%x	按照十六进制形式打印参数
%o	按照八进制形式打印参数

9.1.9 流水线编程

Cambricon BANG C++ 语言为用户提供了丰富的内置接口来进行显式流水线编程。本节介绍 Cambricon BANG C++ 流水线内置接口和并行编程的基本概念，并给出示例。

9.1.9.1 流水线接口

`bang::pipeline` 类提供的接口如下：

```
void wait_compute();
void wait_memcpy();
void wait_copy_nram_to_nram();
void wait_copy_nram_to_dram();
void wait_copy_dram_to_nram();
void wait_copy_wram_to_dram();
```

```
void wait_copy_dram_to_wram();
void wait_copy_sram_to_dram();
void wait_copy_dram_to_sram();
void wait_copy_nram_to_sram();
void wait_copy_sram_to_nram();
void wait_copy_wram_to_sram();
void wait_copy_sram_to_wram();
```

接口 `wait_xx` 等价于 `__sync_xx`。当用 C++ 使用流水线编程，需要增加编译选项 `-std=c++11`。在同步编程模型中，GDRAM->NRAM[read]，NRAM->NRAM[modify]，NRAM->GDRAM[write]，是顺序执行的。在流水线编程模型中，指令是异步执行的。通过重构循环，每个迭代执行的指令属于原始循环的不同迭代。循环重构用于排列循环指令。流水线编程是循环多个迭代并行执行的一种技术。

注意：

在使用 `__memcpy_async` 前插入 `wait_xx`。

9.1.9.2 流水线示例

两级流水线示例如下：

```
// -----
// Async Multi Stage with NRAM
// -----
// | [loop0] pipe0: L0 --> C0 --> S0 | L0 --> C0 --> S0 |
// | [loop0] pipe1: L1 --> C1 --> S1 | L1 --> C1 --> S1 |
// | [loop1] pipe0: L0 --> C0 --> S0 | L0 --> C0 --> S0 |
// | [loop1] pipe1: L1 --> C1 --> S1 | L1 --> C1 --> S1 |
// | [...] |
// -----
__mlu_func__ void blockVecMulMultiStageNMem(float* __restrict__ Cg,
                                             float* __restrict__ Ag,
                                             float* __restrict__ Bg,
                                             float* __restrict__ Cn,
                                             float* __restrict__ An,
                                             float* __restrict__ Bn,
                                             uint32_t sizeC,
                                             bang::pipeline& pipe0,
                                             bang::pipeline& pipe1) {
    // Step 1: Load Ag->An and Bg->Bn, GDRAM->NRAM
    // -----
    bang::memcpy_async(pipe0, An + sizeC * 0,
```

```
Ag + sizeC * taskDim * 0 + sizeC * taskId,
sizeC * sizeof(float), GDRAM2NRAM);
bang::memcpy_async(pipe0, Bn + sizeC * 0,
Bn + sizeC * taskDim * 0 + sizeC * taskId,
sizeC * sizeof(float), GDRAM2NRAM);

// ----- Stage 0: -----
// ----- Stage 1: -----  
Cambricon@155chb

bang::memcpy_async(pipe1, An + sizeC * 1,
Ag + sizeC * taskDim * 1 + sizeC * taskId,
sizeC * sizeof(float), GDRAM2NRAM);
bang::memcpy_async(pipe1, Bn + sizeC * 1,
Bn + sizeC * taskDim * 1 + sizeC * taskId,
sizeC * sizeof(float), GDRAM2NRAM);

// ----- Stage 1: -----
// Step 2: Compute An and Bn
// ----- Stage 0: -----
pipe0.wait_copy_dram_to_nram();
pipe1.wait_copy_dram_to_nram();
__bang_mul(Cn + sizeC * 0, An + sizeC * 0, Bn + sizeC * 0, sizeC);
__bang_mul(Cn + sizeC * 1, An + sizeC * 1, Bn + sizeC * 1, sizeC);
// ----- Stage 0: -----
// Step 3: Store Cn->Cg, NRAM->GDRAM
// ----- Stage 0: -----
pipe0.wait_compute();
pipe1.wait_compute();
bang::memcpy_async(pipe0, Cg + sizeC * taskDim * 0 + sizeC * taskId,
Cn + sizeC * 0, sizeC * sizeof(float), NRAM2GDRAM);

// ----- Stage 0: -----
// ----- Stage 1: -----
bang::memcpy_async(pipe1, Cg + sizeC * taskDim * 1 + sizeC * taskId,
Cn + sizeC * 1, sizeC * sizeof(float), NRAM2GDRAM);
// ----- Stage 1: -----
}
```

9.2 C++ 语言标准支持

9.2.1 数学库函数

下表列出了在设备侧可以使用的标准数学库函数：

表 9.5: Cambricon BANG C 支持的标准数学库函数和宏定义

函数	描述
abs(x)	返回整数类型参数 x 的绝对值
fabsf(x)	返回浮点类型参数 x 的绝对值
acosf(x)	返回浮点类型参数 x 的反余弦值
cosf(x)	返回浮点类型参数 x 的余弦值
coshf(x)	返回浮点类型参数 x 的双曲余弦值
acoshf(x)	返回浮点类型参数 x 的反双曲余弦值
sinf(x)	返回浮点类型参数 x 的正弦值
sinhf(x)	返回浮点类型参数 x 的双曲正弦值
asinf(x)	返回浮点类型参数 x 的反正弦值
asinhf(x)	返回浮点类型参数 x 的反双曲正弦值
tanf(x)	返回浮点类型参数 x 的正切值
atanf(x)	返回浮点类型参数 x 的反正切值
atanhf(x)	返回浮点类型参数 x 的反双曲正切值
atan2f(x, y)	返回两个浮点类型参数 x, y 的反正切值
expf(x)	返回 e^x
expm1f(x)	返回 $e^x - 1$
exp2f(x)	返回 2^x
fmodf(x, y)	返回 x/y 的余数

下页继续

表 9.5 – 续上页

函数	描述
copysignf(x, y)	返回由 x 的值和 y 的符号位构成的浮点数
fmaxf(x, y)	返回两个浮点类型参数 x, y 中较大的一个
fminf(x, y)	返回两个浮点类型参数 x, y 中较小的一个
isnan(x)	判断浮点参数 x 是否是非数
isinf(y)	判断浮点参数 x 是否是无穷数
log2f(x)	返回 $\log_2 x$
log10f(x)	返回 $\log_{10} x$
log1pf(x)	返回 $\ln(1 + x)$
powf(x, y)	返回 x^y
sqrtf(x)	返回 \sqrt{x}
scalbnf(x, y)	返回 $x \times 2^y$
ceilf(x)	返回不小于浮点类型参数 x 的最小整数值
floorf(x)	返回不大于浮点类型参数 x 的最大整数值
truncf(x)	返回绝对值不大于 x 且最接近 x 的整数
roundf(x)	返回浮点参数 x 四舍五入的浮点值

9.2.2 函数指针

Cambricon BANG C 不支持函数指针。

9.2.3 动态初始化

Cambricon BANG C 在设备侧不支持全局变量的动态初始化，仅支持静态初始化。

```
--ldram__ float a;      // OK!
--nram__ int b = f();   // Error!
--nram__ int *c = &b;    // Error!
```

9.2.4 `extern` 修饰符

Cambricon BANG C 不支持使用 `extern` 修饰设备侧的全局变量和函数。

9.3 可配置环境变量

9.3.1 `CNRT_BANGC_PRINTF_LIMIT`

使用 `__bang_printf` 进行调试时，用户可使用环境变量 `CNRT_BANGC_PRINTF_LIMIT` 调整每个 Task 预留的 `printf` buffer 大小，默认为 1024B。如果每个 Task 真实产生的 `printf` 数据超过 `CNRT_BANGC_PRINTF_LIMIT`，则新数据会覆盖原有的数据。

9.4 CNDrv API

本章主要介绍在 CNRT 中涉及到的 CNDrv API 一些概念以及内容，例如 Context、Module 等，以及 CNRT 接口和 CNDrv 接口混合调用的说明。

CNDrv API 是一组更加底层的驱动 API，以 CNDrv 动态库的方式发布。CNDrv API 分为 Device、Context、Module、Queue、Notifier、Memory、Kernel 七部分功能模块，每一部分均提供一系列接口完成该模块的所有功能，同时每一个接口通过一个对应的句柄进行操作。其句柄与功能模块的对应关系如表 9.6 CNDrv 功能模块与句柄对应关系 所示：

表 9.6: CNDrv 功能模块与句柄对应关系

功能模块	功能句柄	功能简述
Device	CNdev	Cambricon 设备
Context	CNcontext	设备侧运行上下文，类似于 CPU 线程
Module	CNmodule	可加载到设备侧的指令与数据包
Queue	CNqueue	执行流
Notifier	CNnotifier	通知机制
Memory	CNaddr	设备侧地址
Kernel	CNkernel	可在设备侧运行的函数

CNDrv API 还包括一个初始化接口 `cnInit()` 和一组错误处理的接口。

- 在使用所有功能模块前，需要通过初始化接口 `cnInit()` 完成 CNDrv 动态库的运行环境初始化；

- 运行时，需要创建 Context 为所有运行时模块提供设备侧运行环境；
- 使用 CNDrv API 接口出错时，可以通过错误处理接口完成对应错误码的获取以及错误原因的获取。

CNDrv API 相比于 CNRT 接口，增加了对设备侧资源控制管理的接口，以达成更高效率的编程模型，同时也增加了编程的复杂度。主要增加的功能管理模块包括 Context 的管理与控制、Module 动态加/卸载、Memory 的细粒度控制功能。

以下代码样例通过下发运行简单的 Kernel 流程来展示基于 CNDrv API 的编程模型：

```
#include "cn_api.h"

int main(int argc, char **argv)
{
    CNdev device;
    CNcontext context;
    CNmodule module;
    CNkernel kernel;
    CNqueue queue;
    CNaddr addr;

    /* Init CNDrv Library */
    ERROR_CHECK(cnInit(0));
    Cambricon@155chb
    /* Get Device 0 Handle */
    ERROR_CHECK(cnDeviceGet(&device, 0));

    /* Create Context For Current Host Thread */
    ERROR_CHECK(cnCtxCreate(&context, 0, device));

    /* Load Module For Current Context */
    ERROR_CHECK(cnModuleLoad("MyModule", &module));

    /* Get Kernel From Module */
    ERROR_CHECK(cnModuleGetKernel(module, "KernelName", &kernel));

    /* Malloc Device Memory */
    ERROR_CHECK(cnMallocNode(&addr, 1024, 2));

    /* Create Queue To Launch Kernel */
    ERROR_CHECK(cnCreateQueue(&queue, 0));

    /* Prepare Kernel Parameters */
    CNaddr params[] = {addr};
```

```

void *extra[] = {
    CN_INVOKE_PARAM_BUFFER_POINTER, (void *)params,
    CN_INVOKE_PARAM_BUFFER_SIZE, (void *)sizeof(params),
    CN_INVOKE_PARAM_END
};

/* InvokeKernel To Device */
ERROR_CHECK(cnInvokeKernel(kernel, 4, 1, 1, CN_KERNEL_CLASS_UNION, 0, queue, NULL, extra));
ERROR_CHECK(cnQueueSync(queue));

/* Destroy All Resource */
ERROR_CHECK(cnFree(addr));
ERROR_CHECK(cnQueryQueue(queue));
ERROR_CHECK(cnModuleUnload(module));
ERROR_CHECK(cnCtxDestroy(context));

return 0;
}

```

上面是一个基础的 CNDrv API 功能使用流程，基本描述出 CNDrv API 的使用流程约束：

1. 使用 cnInit() 初始化 CNDrv 动态库；
2. 通过设备句柄创建 Context，为设备运行提供上下文；
3. 在 Context 的基础上进行内存申请、Queue 创建、Module 加载操作，以及下发执行 Kernel 到设备侧运行；
4. 运行成功后释放所有申请的句柄资源，最终销毁上下文。

9.4.1 Context

Context 类似于一个 CPU 线程，CNDrv 所有的资源以及接口行为都封装在 Context 里。Context 负责管理所有的资源，当 Context 销毁时系统会将所有的资源清空。

Context 分为两种：

- 用户创建的 Context：由用户自行管理，其生命周期开始于调用 cnCtxCreate()，终止于调用 cnCtxDestroy()；对于每个设备，用户可创建若干 Context，其数量取决于系统资源以及 CNDrv 库的限制。
- Shared Context：每个设备下仅唯一存在一个 Shared Context，主要用于多库之间共享 Context 内资源，CNRT 会使用并管理该 Context；Shared Context 仅可以被激活，无法被销毁；其生命周期起始于调用 cnSharedContextAcquire()，终止于调用 cnSharedContextRelease() 或 cnSharedContextReset()。

Context 可以与主机侧用户线程绑定，一个主机侧用户线程可以通过 cnCtxSetCurrent() 绑定唯一一

个 Context，当绑定不同 Context 时当前 Context 将与主机侧线程解绑并与新的 Context 绑定。一个 Context 可以同时被多个主机侧用户线程绑定，但是一个线程同一时刻只能绑定一个 Context。

通过 `cnCtxCreate` 创建 Context 后，将与当前主机侧线程进行绑定。而 Shared Context 在通过 `cnSharedContextAcquire()` 激活后，不会与当前主机侧线程绑定，需要用户显式调用 `cnCtxSetCurrent()` 接口来与当前主机侧线程绑定。假如当前主机侧线程未绑定可用的 Context 或者当前主机侧线程绑定了错误的 Context 时，都将报 `CN_ERROR_INVALID_CONTEXT` 错误。

当主机侧线程绑定 Context 后，后续通过 CNDrv API 接口申请的资源都将属于该 Context，其管理的资源包括：Module、Queue、Kernel、Notifier、Atomic。

9.4.1.1 Context 与设备侧资源之间的关系

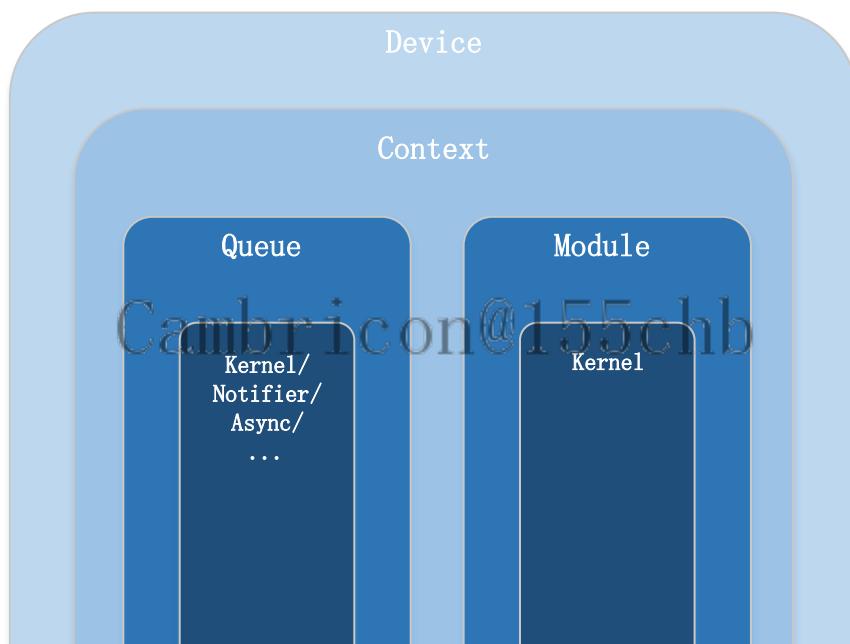


图 9.1: 设备侧资源关系说明

注解：

图中 Queue 内部所列的 Kernel、Notifier、Async 代表可以在 Queue 中执行的任务，其中 Async 代表所有以 Async 为后缀的接口，该接口的任务会被放置于 Queue 中执行。

如图 9.1 设备侧资源关系说明 所示，该图说明所有 CNDrv 可见资源之间的管理关系。从外到内依次包括：

- Device: Device 是指设备，每个设备对应一个 Device 资源，CNDrv 层面仅保留 Device 编号。其生命周期起始于函数 `cnInit()` 的调用，终止于当前进程结束或 CNDrv 动态库的关闭。

- Context: Context 基于 Device 管理，每个 Device 下可以有多个 Context，而每个 Context 则属于唯一的 Device。
- Module: Module 是 Context 内的指令与数据包资源，每个 Context 下存在若干 Module，而某个 Module 则属于唯一的 Context。其生命周期起始于调用 `cnModuleLoad()`，终止于调用 `cnModuleUnload()`。当 Context 销毁时，其内所管理的所有 Module 也不再可用。
- Queue: Queue 是 Context 内的执行流资源，所有异步任务均通过该资源单元执行，Queue 属于唯一的 Context，但一个 Context 下可创建若干 Queue。Queue 分为默认 Queue 与用户创建 Queue。
 - 默认 Queue: 默认 Queue 由 CNDrv 内部默认创建，每个 Context 每个主机侧线程存在一个，且不可被销毁，其生命周期终止于进程结束或 Context 被销毁。
 - 用户创建 Queue: 其生命周期起始于调用 `cnQueueCreate()`，终止于调用 `cnQueueDestroy()`。当 Context 销毁时，该 Context 下的所有 Queue 均不可再用。
- Kernel: Kernel 特指运行在 MLU Core 上的任务，该资源由 Module 管理，每个 Module 下存在多个 Kernel。Kernel 均在 Queue 内执行，但其资源管理者为 Module，故其生命周期同 Module。
- Notifier: Notifier 是用来标记 Queue 执行过程的某个特定的点，Notifier 也可以当成一个操作插入 Queue。主要用途是：同步 Queue 的执行，统计 Kernel 运行时间。其生命周期起始于 `cnCreateNotifier()`，终止于 `cnNotifierDestroy()`。

注意：

- Context 下所管理的资源包括 Queue、Module、Kernel、Notifier、Atomic；
- 所有 Context 下管理的资源，在 Context 销毁后，其资源均不可再用。

9.4.1.2 Context 与主机侧线程之间的关系

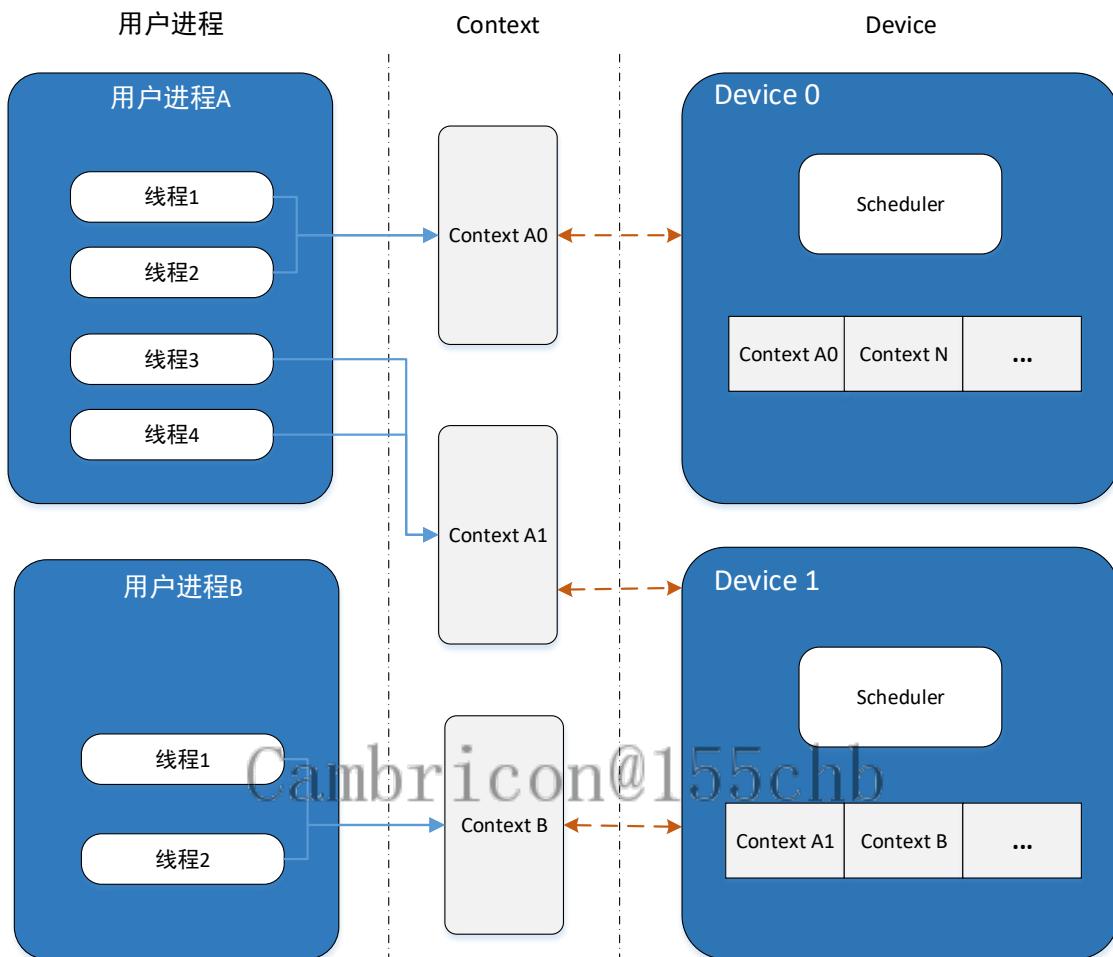


图 9.2: Context 与主机侧线程关系

Context 是设备运行时抽象，为用户提供运行时所需的资源与配置。Context 位于主机侧用户线程与设备侧的中间层，用于提供给用户虚拟的设备运行环境（即 Context）并设置相关配置与参数，同时将用户构造的 Context 内容传递给指定设备，供其调度与运行。如 [图 9.2 Context 与主机侧线程关系](#) 所示，用户进程内的每个线程可绑定一个 Context，而同一个 Context 可同时被多个线程绑定。线程同一时刻仅可绑定一个 Context，所以用户可以通过 `cnCtxSetCurrent()` 做 Context 切换。当新的 Context 需要绑定当前线程时，若当前线程中已有绑定 Context，则原来的 Context 会先与主机侧线程解绑，再与新的 Context 绑定；若当前线程中无绑定的 Context 时，则新的 Context 直接绑定当前主机侧线程。

9.4.2 Module

Module 是由 CNCC (Cambricon Compiler Collection, 寒武纪 Cambricon BANG C 语言编译器) 输出，并可动态加载到设备的指令和数据包，类似于 Windows 中的 dll。

注解：

- 所有的符号的名称包括函数和全局变量都在 Module 范围内维护；
- 在同一个 Context 内，不同的 Module 可以互通。

以下代码演示 Module 的加载，并通过 Module 获取执行 Kernel 的过程，以及在执行完成后卸载 Module：

```
CNmodule cnModule;
CNkernel cnKernel;
CNqueue queue;
void *extra[] = {
    CN_INVOKE_PARAM_BUFFER_POINTER, (void *)params,
    CN_INVOKE_PARAM_BUFFER_SIZE, (void *)sizeof(params),
    CN_INVOKE_PARAM_END
};
/*create a queue*/
cnCreateQueue(&queue, 0);
/*load module*/
cnModuleLoad("cnModule.cnfatbin", &cnModule);
/*get kernel from module */
cnModuleGetKernel(cnModule, "cnKernel", &cnKernel);
/* invoke kernel*/
cnInvokeKernel(cnKernel, 1, 1, 1, CN_KERNEL_CLASS_BLOCK, 0, queue, NULL, extra));
/* wait for kernel to complete*/
cnQueueSync(queue);
/*destroy queue*/
cnDestroyQueue(queue);
/*unload module*/
cnModuleUnload(cnModule);
```

以下代码演示 Module 加载指令内存，并执行指令内存的 Kernel 过程，以及在执行完成后卸载指令内存：

```
CNmodule cnModule;
CNkernel cnKernel;
CNqueue queue;
void *extra[] = {
    CN_INVOKE_PARAM_BUFFER_POINTER, (void *)params,
    CN_INVOKE_PARAM_BUFFER_SIZE, (void *)sizeof(params),
    CN_INVOKE_PARAM_END
```

```
};

char *code_module = "some fatbinary file code";
/*create a queue*/
cnCreateQueue(&queue, 0);
/*load module*/
cnModuleLoadFatBinary(code_module, &cnModule);
/*get kernel from module */
cnModuleGetKernel(cnModule, "cnKernel", &cnKernel);
/* invoke kernel*/
cnInvokeKernel(cnKernel, 1, 1, 1, CN_KERNEL_CLASS_BLOCK, 0, queue, NULL, extra));
/* wait for kernel to complete*/
cnQueueSync(queue);
/*destroy queue*/
cnDestroyQueue(queue);
/*unload module*/
cnModuleUnload(cnModule);
```

注意：

- 当调用 cnModuleLoad() 以及 cnModuleLoadFatbinary() 接口进行 Module 加载时，若没有任何可以在当前 Context 上加载的 Module 时，该接口会返回失败，并返回对应的错误码；
- 当调用 cnModuleGetKernel() 接口时，若 Kernel 对应的 Module 由于当前设备不支持，导致没有被加载，则该接口会由于无法找到对应的 Module 而返回失败，并返回对应的错误码；
- 当前软件版本暂不支持 JIT (Just In Time, 即时编译) 方式进行可执行代码的加载，仅支持通过 Module 文件或内存的方式进行加载。

9.4.3 CNRT 与 CNDrv 的混合使用

同一应用程序内，可以混合使用 CNRT 和 CNDrv 的接口。

当一个 Context 已经被 CNDrv 接口创建并且被设为当前 Context 时，后序的 CNRT 接口则会使用这个 Context 而不是创建新的 Context。

如果 CNRT 已经被初始化（CNRT 隐式初始化请参考 第 6.5.2 章 CNRT 的初始化），那么可以通过调用 cnCtxGetCurrent 接口获取初始化时创建的 Shared Context，这个 Context 可以用于后序的 CNDrv 调用。

设备内存、Queue 和 Notifier 均可以使用 CNRT 或 CNDrv 的接口进行申请、释放、创建、销毁。

```
CNaddr device_ptr;
CNqueue queue;
CNnotifier notifier;
```

```
cnMalloc(&device_ptr);  
cnQueueCreate(&queue, 0);  
cnNotifierCreate(&notifier, 0);  
  
cnrtFree((void *)device_ptr);  
cnrtQueueDestroy((cnrtQueue)queue);  
cnrtNotifierDestroy((cnrtNotifier)notifier);
```

设备管理和版本管理部分的所有接口都可以互换使用。

Cambricon@155chb