Cambricon CNRT Developer Guide

Release 6.7.0

Oct 18, 2023

Table of Contents

1 Copyright

DISCLAIMER

CAMBRICON MAKES NO REPRESENTATION, WARRANTY (EXPRESS, IMPLIED, OR STATUTORY) OR GUARANTEE REGARDING THE INFORMATION CONTAINED HEREIN, AND EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY, TITLE, NONINFRINGEMENT OF INTELLECTUAL PROPERTY OR FITNESS FOR A PARTICULAR PURPOSE, AND CAMBRICON DOES NOT ASSUME ANY LIABILITY ARISING OUT OF THE APPLICATION OR USE OF ANY PRODUCT OR SERVICES. CAMBRICON SHALL HAVE NO LIABILITY RELATED TO ANY DEFAULTS, DAMAGES, COSTS OR PROBLEMS WHICH MAY BE BASED ON OR ATTRIBUTABLE TO: (I) THE USE OF THE CAMBRICON PRODUCT IN ANY MANNER THAT IS CONTRARY TO THIS GUIDE, OR (II) CUSTOMER PRODUCT DESIGNS.

LIMITATION OF LIABILITY

In no event shall Cambricon be liable for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption and loss of information) arising out of the use of or inability to use this guide, even if Cambricon has been advised of the possibility of such damages. Notwithstanding any damages that customer might incur for any reason whatsoever, Cambricon's aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the Cambricon terms and conditions of sale for the product.

ACCURACY OF INFORMATION

Information provided in this document is proprietary to Cambricon, and Cambricon reserves the right to make any changes to the information in this document or to any products and services at any time without notice. The information contained in this guide and all other information contained in Cambricon documentation referenced in this guide is provided "AS IS." Cambricon does not warrant the accuracy or completeness of the information, text, graphics, links or other items contained within this guide. Cambricon may make changes to this guide, or to the products described therein, at any time without notice, but makes no commitment to update this guide.

Performance tests and ratings set forth in this guide are measured using specific chips or computer systems or components. The results shown in this guide reflect approximate performance of Cambricon products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. As set forth above, Cambricon makes no representation, warranty or guarantee that the product described in this guide will be suitable for any specified use. Cambricon does not represent or warrant that it tests all parameters of each product. It is customer's sole responsibility to ensure that the product is suitable and fit for the application planned by the customer and to do the necessary testing for the application in order to avoid a default of the application or the product.

Weaknesses in customer's product designs may affect the quality and reliability of Cambricon product and may result in additional or different conditions and/or requirements beyond those contained in this guide.

IP NOTICES

Cambricon and the Cambricon logo are trademarks and/or registered trademarks of Cambricon Corporation in China and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

This guide is copyrighted and is protected by worldwide copyright laws and treaty provisions. This guide may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without Cambricon's prior written permission. Other than the right for customer to use the information in this guide with the product, no other right or license, either express or implied, is granted by Cambricon under this guide. For the avoidance of doubt, Cambricon does not grant any right or license (express or implied) to customer under any patents, copyrights, trademarks, trade secret or any other intellectual property or proprietary rights of Cambricon.

- Copyright
- © 2023 Cambricon Corporation. All rights reserved.

2 Update History

This section lists content changes on documentation that were made for each product release.

- V6.7.0

  Date: Aug 15, 2023

  Changes:

  - Added the following new enum members in cnrtDeviceAttr_t:
    - cnrtAttrIPCNotifierSupported
- V6.6.0

  Date: Jun 15, 2023

  Changes:

  - Added the following new members in cnrtDeviceProp_t:
    - cnrtUUID_t uuid
  - Added the following new APIs:
    - cnrtMallocConstant
    - cnrtMemcpyAsync_V2
    - cnrtMemcpyFromSymbolAsync_V2
    - cnrtMemcpyToSymbolAsync_V2
- V6.5.0

  Date: May 01, 2023

  Changes:

  - Added the following new enum members in cnrtDeviceAttr_t:
    - cnrtAttrMDRMemorySize
    - cnrtAttrAvailableGlobalMemorySize
  - Added the following new enum members in cnrtDeviceProp_t
    - MDRMemorySize
    - availableGlobalMemorySize
  - Added the following new APIs:
    - cnrtGetSymbolAddress
    - cnrtGetSymbolSize
    - cnrtMemcpyFromSymbol
    - cnrtMemcpyToSymbol

    * cnrtMemcpyFromSymbolAsync
    * cnrtMemcpyToSymbolAsync

- V6.4.0

  Date: Dec 29, 2022

  Changes:

  – Added the following new enum members in cnrtDeviceAttr_t:
    * cnrtAttrAIIspCoreSupported
    * cnrtAttrMultiCtxNotifierWaitSupported

  – Added the following new API:
    * cnrtQueueGetPriority

- V6.3.0

  Date: Dec 6, 2022

  Changes:

  – Added the following new enum member in cnrtDeviceAttr_t:
    * cnrtAttrComputeMode

  – Added the following new enum:
    * cnrtComputeMode_t

  – Added the following new enum members in cnrtDeviceProp_t:
    * totalMem
    * maxDim[3]
    * ipuClockRate
    * memClockRate
    * totalConstMem
    * major
    * minor
    * ECCEnabled
    * pciBusID
    * pciDeviceID
    * pciDomainID
    * maxL2CacheSize
    * persistingL2CacheMaxSize
    * queuePrioritiesSupported
    * sparseComputingSupported
    * FP16ComputingSupported
    * INT8ComputingSupported
    * BF16ComputingSupported
    * TF32ComputingSupported
    * maxQueueSize
    * maxNotifierSize

    * tinyCoreSupported
    * codecJPEGSupported
    * codecH264Supported
    * codecH265Supported
    * maxClusterCountPerUnionTask
    * clusterCount
    * McorePerCluster
    * maxQuadrantCount
    * maxUnionTypePerQuadrant
    * maxClusterPerUnionLimitTask
    * ISAVersion
    * isMultipleTensorProcessor
    * NramSizePerMcore
    * WramSizePerMcore
    * LmemSizePerMcore
    * SramSizePerMcore
    * globalMemoryNodeCoun
    * cacheSize
    * GmemBusWidth
    * computeMode

- V6.2.0

  Date: Oct 13, 2022

  Changes:

  – Added the following new enum member in cnrtRet_t:

    * cnrtErrorKernelTrap

- V6.0.0

  Date: Jul 8, 2022

  Changes:

  – Added the following new APIs:

    * cnrtQueueSetAttribute
    * cnrtQueueGetAttribute
    * cnrtQueueCopyAttributes
    * cnrtQueueBeginCapture
    * cnrtQueueEndCapture
    * cnrtQueueIsCapturing
    * cnrtQueueGetCaptureInfo
    * cnrtQueueUpdateCaptureDependencies
    * cnrtTaskTopoCreate
    * cnrtTaskTopoDestroy

* cnrtTaskTopoClone
* cnrtTaskTopoNodeFindInClone
* cnrtTaskTopoDestroyNode
* cnrtTaskTopoGetEdges
* cnrtTaskTopoGetNodes
* cnrtTaskTopoGetRootNodes
* cnrtTaskTopoAddDependencies
* cnrtTaskTopoRemoveDependencies
* cnrtTaskTopoNodeGetDependencies
* cnrtTaskTopoNodeGetDependentNodes
* cnrtUserObjectCreate
* cnrtUserObjectAcquire
* cnrtUserObjectRelease
* cnrtTaskTopoAcquireUserObject
* cnrtTaskTopoReleaseUserObject
* cnrtTaskTopoAddEmptyNode
* cnrtTaskTopoAddHostNode
* cnrtTaskTopoHostNodeGetParams
* cnrtTaskTopoHostNodeSetParams
* cnrtTaskTopoAddKernelNode
* cnrtTaskTopoKernelNodeGetParams
* cnrtTaskTopoKernelNodeSetParams
* cnrtTaskTopoAddMemcpyNode
* cnrtTaskTopoMemcpyNodeGetParams
* cnrtTaskTopoMemcpyNodeSetParams
* cnrtTaskTopoAdd:MemsetNode
* cnrtTaskTopoMemsetNodeGetParams
* cnrtTaskTopoMemsetNodeSetParams
* cnrtTaskTopoAddChildTopoNode
* cnrtTaskTopoChildTopoNodeGetTopo
* cnrtTaskTopoInstantiate
* cnrtTaskTopoEntityDestroy
* cnrtTaskTopoEntityInvoke
* cnrtTaskTopoDebugDotPrint
* cnrtTaskTopoKernelNodeGetAttribute
* cnrtTaskTopoKernelNodeSetAttribute
* cnrtTaskTopoKernelNodeCopyAttributes
* cnrtTaskTopoEntityHostNodeSetParams
* cnrtTaskTopoEntityKernelNodeSetParams

    * cnrtTaskTopoEntityMemcpyNodeSetParams
    * cnrtTaskTopoEntityMemsetNodeSetParams
    * cnrtTaskTopoEntityChildTopoNodeSetParams
    * cnrtTaskTopoEntityUpdate

  - Added the following new enums:
    * cnrtAccessProperty_t
    * cnrtQueueAttrID_t
    * cnrtQueueCaptureStatus_t
    * cnrtQueueCaptureMode_t
    * cnrtUpdateQueueCaptureDependenciesFlags_t
    * cnrtUserObjectAcquireflags_t
    * cnrtTaskTopoNodeType_t
    * cnrtKernelNodeAttr_t
    * cnrtTaskTopoDebugDotFlags_t
  - Added the following new unions:
    * cnrtQueueAttrValue_t
    * cnrtKernelNodeAttrValue_t
  - Added the following new structs:
    * cnrtAccessPolicyWindow_t
    * cnrtHostNodeParams_t
    * cnrtKernelNodeParams_t
    * cnrtMemsetParams_t
    * cnrtTaskTopo_t
    * cnrtTaskTopoNode_t
    * cnrtUserObject_t
    * cnrtTaskTopoEntity_t
  - Added the following new enum members in cnrtRet_t:
    * cnrtErrorIllegalState
    * cnrtErrorSysNoMem
    * cnrtErrorQueueCaptureUnsupported
    * cnrtErrorQueueCaptureInvalidated
    * cnrtErrorQueueCaptureWrongThread
  - Added the following new enum member in cnrtDeviceAttr_t:
    * cnrtAttrMaxPersistingL2CacheSize
  - Removed the following APIs:
    * cnrtVBInit
    * cnrtVBExit
    * cnrtVBSetComCfg
    * cnrtVBGetComCfg

    - cnrtVBAllocBlock
    - cnrtVBFreeBlock
    - cnrtVBBlockRefInc
    - cnrtVBBlockRefDec
    - cnrtVBPhy2Handle
    - cnrtVBHandle2Phy
    - cnrtVBHandle2PoolId
    - cnrtVBMmapPool
    - cnrtVBMunmapPool
    - cnrtVBGetBlkUva
    - cnrtVBPutBlkUva
    - cnrtVBInitMod
    - cnrtVBExitMod
    - cnrtVBSetModCfg
    - cnrtVBGetModCfg
    - cnrtVBCreatePool
    - cnrtVBCreatePoolV1
    - cnrtVBDestroyPool
    - cnrtVBGetStat
    - cnrtVBSetUserMeta
    - cnrtVBGetUserMeta
    - cnrtVBSetUserMetaCfg
    - cnrtVBCacheOperation
    - cnrtMallocExt
    - cnrtMallocExtCached

- V5.8.0

  Date: Feb 28, 2022

  Changes:

  - Added the following new APIs:
    - cnrtNotifierCreateWithFlags
    - cnrtDeviceGetQueuePriorityRange
    - cnrtQueueCreateWithPriority
    - cnrtInvokeHostFunc
    - cnrtDeviceGetConfig
    - cnrtDeviceSetConfig
  - Added the following new enums:
    - cnrtNotifierFlags_t
    - cnrtDeviceConfig_t
  - Added the following new enum members in cnrtRoundingMode_t:

    - cnrtRounding_ro
    - cnrtRounding_rm
  - Added the following new enum members in cnrtDeviceFlags_t:
    - cnrtDeviceScheduleAuto
  - Added the following new types:
    - cnrtHostFn_t
  - Added the following new error code：
    - cnrtErrorOpsNotPermitted

- V5.7.0

  Date: Dec 31, 2021

  Changes:

  - Added the following new enum members in cnrtPointerAttributes_t:
    - cacheMode
    - deviceBasePointer
  - Added the following new error code：
    - cnrtErrorInvalidResourceHandle
  - Added the following new API:
    - cnrtVBCreatePoolV1

- V5.6.0

  Date: Oct 19, 2021

  Changes:

  - Added the following new enum members in cnrtDeviceAttr_t:
    - cnrtAttrMaxQuadrantCount
    - cnrtAttrMaxUnionTypePerQuadrant
    - cnrtAttrMaxClusterPerUnionLimitTask
    - cnrtAttrISAVersion
    - cnrtAttrIsMultipleTensorProcessor
    - cnrtAttrTinyCoreSupported
    - cnrtAttrCodecJPEGSupported
    - cnrtAttrCodecH264Supported
    - cnrtAttrCodecH265Supported
    - cnrtAttrMaxClusterCountPerUnionTask
    - cnrtAttrMaxL2CacheSize
    - cnrtAttrTotalConstMemorySize
    - cnrtAttrGlobalMemoryNodeCount
  - Added the following new enum types:
    - cnrtVBUID_t
    - cnrtVBRemapMode_t
  - Added the following new types:

    * cnrtVBPoolConfigs_t
    * cnrtVBConfigs_t
    * cnrtVBModConfigs_t
    * cnrtPos_t
    * cnrtPitchedPtr_t
    * cnrtExtent_t
    * cnrtMemcpy3dParam_t
  - Added the following new APIs:
    * cnrtMallocExt
    * cnrtMallocExtCached
    * cnrtMcacheOperation
    * cnrtMmap
    * cnrtMmapCached
    * cnrtMunmap
    * cnrtVBInit
    * cnrtVBExit
    * cnrtVBSetComCfg
    * cnrtVBGetComCfg
    * cnrtVBAllocBlock
    * cnrtVBFreeBlock
    * cnrtVBBlockRefInc
    * cnrtVBBlockRefDec
    * cnrtVBPhy2Handle
    * cnrtVBHandle2Phy
    * cnrtVBHandle2PoolId
    * cnrtVBMmapPool
    * cnrtVBMunmapPool
    * cnrtVBGetBlkUva
    * cnrtVBPutBlkUva
    * cnrtVBInitMod
    * cnrtVBExitMod
    * cnrtVBSetModCfg
    * cnrtVBGetModCfg
    * cnrtVBCreatePool
    * cnrtVBDestroyPool
    * cnrtVBGetStat
    * cnrtVBSetUserMeta
    * cnrtVBGetUserMeta
    * cnrtVBSetUserMetaCfg

    * cnrtVBCacheOperation
    * cnrtMemcpy2D
    * cnrtMemcpy3D

- V5.5.0

  Date: Aug 30, 2021

  Changes:

  - Added the following new API:
    * cnrtCastDataType_V2
  - Added the following new enums:
    * cnrtDataType_V2_t
    * cnrtRoundingMode_t

- V5.2.0

  Date: May 14, 2021

  Changes:

  - Added the following new APIs:
    * cnrtDeviceGetPCIBusId
    * cnrtDeviceGetByPCIBusId

- V5.0.1

  Date: March 31, 2021

  Changes:

  - Changed the time unit parameter of cnrtNotifierDurationChanged API from millisecond (ms) to microsecond (us).

- V5.0.0

  Date: March 20, 2021

  Changes:

  - Added the following new APIs:
    * cnrtDeviceGetAttribute
    * cnrtGetDeviceProperties
    * cnrtGetDevice
    * cnrtSetDevice
    * cnrtDeviceReset
    * cnrtHostMalloc
    * cnrtMemGetInfo
    * cnrtPointerGetAttributes
    * cnrtNotifierCreate
    * cnrtNotifierDestroy
    * cnrtNotifierElapsedTime
    * cnrtQueueCreate
    * cnrtQueueDestroy

    - cnrtQueueQuery
    - cnrtQueueSync
    - cnrtDriverGetVersion
    - cnrtGetLastError
    - cnrtPeekAtLastError
- Deleted the following enums:
    - cnrt_queue_sync_type
    - cnrtChannelType_t
    - cnrtMallocExType_t
    - cnrtCacheOps_t
    - cnrtInvokeFuncParam_t
    - cnrtInvokeParamType_t
    - cnrtClusterAffinity_t
    - cnrtInvokeParam_t
- Modified the following enums:
    - cnrtRet_t
    - cnrtFunctionType_t
    - cnrtJobType_t
    - cnrtMemTransDir_t
    - cnrtMemType_t
    - cnrtDataType_t
- Added the following enums:
    - cnrtPointerAttributes_t
    - cnrtDeviceAttr_t
    - cnrtDeviceP2PAttr_t
    - cnrtDeviceProp_t

3 Data Type Reference

3.1 enum cnrtRet_t

```
typedef enum {
    cnrtSuccess = 0,
    cnrtErrorNotReady = 1,
    cnrtErrorInit = 100002,
    cnrtErrorNoDevice = 100004,
    cnrtErrorDeviceInvalid = 100005,
    cnrtErrorArgsInvalid = 100006,
    cnrtErrorSys = 100007,
    cnrtErrorSysNoMem = 100010,
    cnrtErrorInvalidResourceHandle = 100014,
    cnrtErrorIllegalState = 100015,
    cnrtErrorNotSupport = 100050,
    cnrtErrorOpsNotPermitted = 100051,
    cnrtErrorQueue = 100060,
    cnrtErrorNoMem = 100100,
    cnrtErrorAssert = 100128,
    cnrtErrorKernelTrap = 100132,
    cnrtErrorKernelUserTrap = 100133,
    cnrtErrorNotFound = 100301,
    cnrtErrorInvalidKernel = 100302,
    cnrtErrorNoKernel = 101312,
```

cnrtErrorNoModule = 101315,

cnrtErrorQueueCaptureUnsupported = 100360,

cnrtErrorQueueCaptureInvalidated = 100361,

cnrtErrorQueueCaptureWrongThread = 100362,

cnrtErrorQueueCaptureMerged = 100363,

cnrtErrorQueueCaptureUnjoined = 100364,

cnrtErrorQueueCaptureIsolation = 100365,

cnrtErrorQueueCaptureUnmatched = 100366,

cnrtErrorTaskTopoEntityUpdateFailure = 100400,

cnrtErrorSetOnActiveProcess = 632002,

cnrtErrorDevice = 632006,

cnrtErrorNoAttribute = 632009,

cnrtErrorMemcpyDirectionInvalid = 632013,

cnrtErrorBusy = 632014,

cnrtErrorCndrvFuncCall = 632015,

cnrtErrorCndevFuncCall = 632017,

cnrtErrorNoCnrtContext = 632019,

cnrtErrorCndrvFuncNotFound = 632020,

cnrtErrorInvalidSymbol = 632100,

cnrtErrorUnknown = 999991,

cnrtErrorMax,

} cnrtRet_t;

`enum cnrtRet_t`

Describes the return values of CNRT APIs.

Values:

`enumerator cnrtSuccess`

The API call returns with no errors.

`enumerator cnrtErrorNotReady`

This indicates that the device or resource requested is busy now.

`enumerator cnrtErrorInit`

This indicates that initialization of CNRT fails.

`enumerator cnrtErrorNoDevice`

This indicates that no device is found.

`enumerator cnrtErrorDeviceInvalid`

This indicates that the device ordinal passed to the API is out of range [0, *cnrtGetDeviceCount()* - 1].

`enumerator cnrtErrorArgsInvalid`

This indicates that one of the parameters passed to the API is invalid or out of range.

`enumerator cnrtErrorSys`

This indicates that it fails to call system API.

`enumerator cnrtErrorSysNoMem`

This indicates that there is insufficient host memory.

`enumerator cnrtErrorInvalidResourceHandle`

This indicates that resource handle passed to the API is invalid.

`enumerator cnrtErrorIllegalState`

This indicates that a resource required by the API is not in a valid state to perform the request operation.

`enumerator cnrtErrorNotSupport`

This indicates that the feature is not supported now.

`enumerator cnrtErrorOpsNotPermitted`

This indicates that the attempted operation is not permitted.

`enumerator cnrtErrorQueue`

This indicates that it fails to get the default queue.

`enumerator cnrtErrorNoMem`

This indicates that there is insufficient MLU device memory.

`enumerator cnrtErrorAssert`

This indicates that a device-side assert is triggered during kernel execution.

`enumerator cnrtErrorKernelTrap`

This indicates that a device-side code proceeds to a specific trap.

`enumerator cnrtErrorKernelUserTrap`

This indicates that a device-side code proceeds to a user specific trap.

`enumerator cnrtErrorNotFound`

This indicates that specific resources are not found.

`enumerator cnrtErrorInvalidKernel`

This indicates that the kernel handle is invalid.

`enumerator cnrtErrorNoKernel`

This indicates that CNCC has not registered the kernel to CNRT.

`enumerator cnrtErrorNoModule`

This indicates that CNCC has not registered the module to CNRT or fails to load the module.

`enumerator cnrtErrorQueueCaptureUnsupported`

This indicates that the operation is not permitted when the queue is capturing.

`enumerator cnrtErrorQueueCaptureInvalidated`

This indicates that the current capture sequence on the queue has been invalidated due to previous error.

`enumerator cnrtErrorQueueCaptureWrongThread`

This indicates that the current capture sequence on the queue does not begin with *cnrtQueueCaptureModeRelaxed* mode, and ends in a different thread.

`enumerator cnrtErrorQueueCaptureMerged`

This indicates that two independent capture sequences are merged.

`enumerator cnrtErrorQueueCaptureUnjoined`

This indicates that the capture sequence contains at least a fork that is not joined to the primary queue.

`enumerator cnrtErrorQueueCaptureIsolation`

This indicates that a queue in capture sequence is trying to create a dependency which crosses the queue capture boundary.

`enumerator cnrtErrorQueueCaptureUnmatched`

This indicates that the queue in capture sequence is not the initially captured one.

`enumerator cnrtErrorTaskTopoEntityUpdateFailure`

This indicates that the Task Topo update is not performed because it includes changes which violate constraints specific to Task Topo entity update.

`enumerator cnrtErrorSetOnActiveProcess`

This indicates that it fails to set device flag because the process is still active.

`enumerator cnrtErrorDevice`

This indicates that current resource is not from the current device.

`enumerator cnrtErrorNoAttribute`

This indicates that the attribute queried does not exist.

`enumerator cnrtErrorMemcpyDirectionInvalid`

This indicates that memory copying direction passed to the API is not in *cnrtMemTransDir_t*.

`enumerator cnrtErrorBusy`

Deprecated. This indicates that the device or resource is busy.

`enumerator cnrtErrorCndrvFuncCall`

This indicates that it fails to call CNDrv API.

`enumerator cnrtErrorCndevFuncCall`

This indicates that it fails to call CNDev API.

`enumerator cnrtErrorNoCnrtContext`

This indicates that no CNRT Context is found.

`enumerator cnrtErrorCndrvFuncNotFound`

This indicates that CNDrv API is not found.

`enumerator cnrtErrorInvalidSymbol`

This indicates that the symbol name/identifier passed to the API is invalid.

`enumerator cnrtErrorUnknown`

Unknown error.

`enumerator cnrtErrorMax`

The last one.

3.2 enum cnrtFunctionType_t

typedef enum {

cnrtFuncTypeBlock = 1,

cnrtFuncTypeBlock0 = cnrtFuncTypeBlock,

cnrtFuncTypeBlock1 = cnrtFuncTypeBlock0 + 1,

cnrtFuncTypeUnion1 = 4,

cnrtFuncTypeUnion2 = 8,

cnrtFuncTypeUnion4 = 16,

cnrtFuncTypeUnion8 = 32,

cnrtFuncTypeUnion16 = 64,

cnrtFuncTypeMutable = - 1,

cnrtJobTypeBlock = cnrtFuncTypeBlock,

cnrtJobTypeUnion1 = cnrtFuncTypeUnion1,

cnrtJobTypeUnion2 = cnrtFuncTypeUnion2,

cnrtJobTypeUnion4 = cnrtFuncTypeUnion4,

} cnrtFunctionType_t;

`enum cnrtFunctionType_t`

Describes the number of cores used for computation on the MLU devices.

Values:

`enumerator cnrtFuncTypeBlock`

One MLU core is used to execute tasks.

`enumerator cnrtFuncTypeBlock0`

The IP core 0 is used to execute tasks.

`enumerator cnrtFuncTypeBlock1`

The IP heterogeneous core 1 is used to execute tasks.

`enumerator cnrtFuncTypeUnion1`

Four MLU cores are used to execute tasks.

`enumerator cnrtFuncTypeUnion2`

Eight MLU cores are used to execute tasks.

`enumerator cnrtFuncTypeUnion4`

16 MLU cores are used to execute tasks.

`enumerator cnrtFuncTypeUnion8`

32 MLU cores are used to execute tasks.

`enumerator cnrtFuncTypeUnion16`

64 MLU cores are used to execute tasks.

`enumerator cnrtFuncTypeMutable`

Not used now.

`enumerator cnrtJobTypeBlock`

One MLU core is used to execute tasks. This is only used for tensor dimension mutable function.

`enumerator cnrtJobTypeUnion1`

Four MLU cores are used to execute tasks. This is only used for tensor dimension mutable function.

`enumerator cnrtJobTypeUnion2`

Eight MLU cores are used to execute tasks. This is only used for tensor dimension mutable function.

`enumerator cnrtJobTypeUnion4`

16 MLU cores are used to execute tasks. This is only used for tensor dimension mutable function.

3.3 enum cnrtComputeMode_t

typedef enum {

cnrtComputeModeDefault = 0,

cnrtComputeModeExclusiveProcess = 1,

} cnrtComputeMode_t;

`enum cnrtComputeMode_t`

Describes the compute modes of an MLU device.

Values:

`enumerator cnrtComputeModeDefault`

Default compute mode: multiple threads can use *cnrtSetDevice()* with this device.

`enumerator cnrtComputeModeExclusiveProcess`

Compute-exclusive-process mode: many threads in one process will be able to use *cnrtSetDevice()* with this device .

3.4 enum cnrtDeviceAttr_t

typedef enum {

cnrtAttrComputeCapabilityMajor = 0x01,

cnrtAttrComputeCapabilityMinor = 0x02,

cnrtAttrSparseComputingSupported = 0x03,

cnrtAttrFP16ComputingSupported = 0x04,

cnrtAttrINT4ComputingSupported = 0x05,

cnrtAttrINT8ComputingSupported = 0x06,

cnrtAttrBF16ComputingSupported = 0x07,

cnrtAttrTF32ComputingSupported = 0x08,

cnrtAttrComputeMode = 0x09,

cnrtAttrQueueSize = 0x101,

cnrtAttrNotifierSize = 0x102,

cnrtAttrSupportQueuePriorities = 0x103,

cnrtAttrTinyCoreSupported = 0x104,

cnrtAttrCodecJPEGSupported = 0x105,

cnrtAttrCodecH264Supported = 0x106,

cnrtAttrCodecH265Supported = 0x107,

cnrtAttrAIIspCoreSupported = 0x108,

cnrtAttrMultiCtxNotifierWaitSupported = 0x109,

cnrtAttrIPCNotifierSupported = 0x10a,

cnrtAttrMaxDimX = 0x201,

cnrtAttrMaxDimY = 0x202,

cnrtAttrMaxDimZ = 0x203,

cnrtAttrMaxClusterCountPerUnionTask = 0x204,

cnrtAttrClusterCount = 0x205,

cnrtAttrMcorePerCluster = 0x206,

cnrtAttrMaxQuadrantCount = 0x207,

cnrtAttrMaxUnionTypePerQuadrant = 0x208,

cnrtAttrMaxClusterPerUnionLimitTask = 0x209,

cnrtAttrISAVersion = 0x20a,

cnrtAttrIsMultipleTensorProcessor = 0x20b,

cnrtAttrMaxL2CacheSize = 0x301,

cnrtAttrNramSizePerMcore = 0x302,

cnrtAttrWramSizePerMcore = 0x303,

cnrtAttrTotalConstMemorySize = 0x304,

cnrtAttrLmemSizePerMcore = 0x305,

cnrtAttrSramSizePerMcore = 0x306,

cnrtAttrGlobalMemoryNodeCount = 0x307,

cnrtAttrMaxPersistingL2CacheSize = 0x309,

cnrtAttrAvailableGlobalMemorySize = 0x312,

cnrtAttrEccEnable = 0x401,

cnrtAttrIpuClockRate = 0x402,

cnrtAttrMemClockRate = 0x403,

cnrtAttrGmemBusWidth = 0x404,

cnrtAttrTotalMemSize = 0x405,

cnrtAttrPciBusID = 0x406,

cnrtAttrPciDeviceID = 0x407,

cnrtAttrPciDomainID = 0x408,

cnrtAttrMDRMemorySize = 0x409,

cnrtAttrUnsupportedFlag = 0xffffff,

cnrtAttrCanMapHostMemory,

cnrtAttrCanSetQueueSize,

cnrtAttrCanSetNotifierSize,

cnrtAttrConcurrentKernels,

cnrtAttrSupportUnifiedAddr,

cnrtAttrSupportManagedMem,

cnrtAttrSupportNativeAtomic,

cnrtAttrSupportPageableMemAccess,

cnrtAttrCanUseHostPointer,

cnrtAttrSupportHostRegsiter,

cnrtAttrCacheSize,

cnrtAttrMaxNum,

} cnrtDeviceAttr_t;

`enum cnrtDeviceAttr_t`

Describes the attributes of the MLU device.

Values:

`enumerator cnrtAttrComputeCapabilityMajor`

Major compute capability of the MLU device.

`enumerator cnrtAttrComputeCapabilityMinor`

Minor compute capability of the MLU device.

`enumerator cnrtAttrSparseComputingSupported`

1: The device supports sparse computing; 0: The device does not.

`enumerator cnrtAttrFP16ComputingSupported`

1: The device supports FP16; 0: The device does not.

`enumerator cnrtAttrINT4ComputingSupported`

1: The device supports INT4; 0: The device does not.

`enumerator cnrtAttrINT8ComputingSupported`

1: The device supports INT8; 0: The device does not.

`enumerator cnrtAttrBF16ComputingSupported`

1: The device supports BF16; 0: The device does not.

`enumerator cnrtAttrTF32ComputingSupported`

1: The device supports TF32; 0: The device does not.

`enumerator cnrtAttrComputeMode`

The compute mode that the device is currently in. See *cnrtComputeMode_t* for details.

`enumerator cnrtAttrQueueSize`

The maximum number of queues.

`enumerator cnrtAttrNotifierSize`

The maximum number of notifiers.

`enumerator cnrtAttrSupportQueuePriorities`

1: The device supports setting queue priorities; 0: The device does not.

`enumerator cnrtAttrTinyCoreSupported`

1: The device supports using tiny core to accelerate collective inter-device or intra-device communication; 0: The device does not.

`enumerator cnrtAttrCodecJPEGSupported`

1: The device supports hardware JPEG codec acceleration; 0: The device does not.

`enumerator cnrtAttrCodecH264Supported`

1: The device supports hardware video H.264 codec acceleration; 0: The device does not.

`enumerator cnrtAttrCodecH265Supported`

1: The device supports hardware video H.265 codec acceleration; 0: The device does not.

`enumerator cnrtAttrAIIspCoreSupported`

1: The device supports AI ISP core. 0: The device does not.

`enumerator cnrtAttrMultiCtxNotifierWaitSupported`

The device supports wait notifier on another context's queue.

`enumerator cnrtAttrIPCNotifierSupported`

The device supports ipcnotifier functions via *cnrtIpcGetNotifierHandle* and *cnrtIpcOpenNotifierHandle*.

`enumerator cnrtAttrMaxDimX`

The maximum block dimension X.

`enumerator cnrtAttrMaxDimY`

The maximum block dimension Y.

`enumerator cnrtAttrMaxDimZ`

The maximum block dimension Z.

`enumerator cnrtAttrMaxClusterCountPerUnionTask`

The maximum number of clusters per union task.

`enumerator cnrtAttrClusterCount`

The maximum number of clusters of the MLU device.

`enumerator cnrtAttrMcorePerCluster`

The maximum number of MLU cores of each cluster.

`enumerator cnrtAttrMaxQuadrantCount`

The maximum count of quadrants per device. Intra-quadrant clusters have the best unified memory access performance.

`enumerator cnrtAttrMaxUnionTypePerQuadrant`

The maximum union task types that can maintain unified intra-quadrant memory access.

`enumerator cnrtAttrMaxClusterPerUnionLimitTask`

The maximum number of clusters per union limitation task.

`enumerator cnrtAttrISAVersion`

ISA version of current MLU device in the form of three-digit number.

`enumerator cnrtAttrIsMultipleTensorProcessor`

1: The device adopts multi-tensor-processor architecture; 0: The device does not.

`enumerator cnrtAttrMaxL2CacheSize`

The size of L2 cache in bytes.

`enumerator cnrtAttrNramSizePerMcore`

The maximum NRAM memory available of each MLU core in bytes.

`enumerator cnrtAttrWramSizePerMcore`

The maximum WRAM memory available of each MLU core in bytes.

`enumerator cnrtAttrTotalConstMemorySize`

The memory available on device for **mlu_const** variable in a Cambricon BANG C kernel in MB.

`enumerator cnrtAttrLmemSizePerMcore`

The maximum local memory available of each core in MB.

`enumerator cnrtAttrSramSizePerMcore`

The maximum SRAM memory available of each cluster in bytes.

`enumerator cnrtAttrGlobalMemoryNodeCount`

The number of NUMA nodes on device.

`enumerator cnrtAttrMaxPersistingL2CacheSize`

The maximum L2 persisting cache size in bytes.

`enumerator cnrtAttrAvailableGlobalMemorySize`

Available global memory size in MB.

`enumerator cnrtAttrEccEnable`

1: The device supports ECC; 0: The device does not.

`enumerator cnrtAttrIpuClockRate`

The cluster clock frequency in kilohertz.

`enumerator cnrtAttrMemClockRate`

The memory clock frequency in kilohertz.

`enumerator cnrtAttrGmemBusWidth`

The global memory bus width in bits.

`enumerator cnrtAttrTotalMemSize`

The maximum available memory in megabytes.

`enumerator cnrtAttrPciBusID`

The PCI bus identifier of the MLU device.

`enumerator cnrtAttrPciDeviceID`

The PCI device identifier of the MLU device.

`enumerator cnrtAttrPciDomainID`

The PCI domain ID of the MLU device.

`enumerator cnrtAttrMDRMemorySize`

MDR memory size in megabytes. Not supported yet.

`enumerator cnrtAttrUnsupportedFlag`

Not supported flag.

`enumerator cnrtAttrCanMapHostMemory`

1: The device supports mapping host memory to MLU; 0: The device does not.

`enumerator cnrtAttrCanSetQueueSize`

1: The device supports setting the maximum queue size; 0: The device does not.

`enumerator cnrtAttrCanSetNotifierSize`

1: The device supports setting the maximum notifier size; 0: The device does not.

`enumerator cnrtAttrConcurrentKernels`

1: The device supports multiple kernels within the same Context simultaneously; 0: The device does not.

`enumerator cnrtAttrSupportUnifiedAddr`

1: The device supports sharing a unified address space with the host; 0: The device does not.

`enumerator cnrtAttrSupportManagedMem`

1: The device supports allocating managed memory; 0: The device does not.

`enumerator cnrtAttrSupportNativeAtomic`

1: The link between the device and the host supports native atomic operations; 0: The link between the device and the host does not.

`enumerator cnrtAttrSupportPageableMemAccess`

1: The device supports accessing pageable memory coherently; 0: The device does not.

`enumerator cnrtAttrCanUseHostPointer`

1: The device can access host registered memory at the same virtual address as the CPU; 0: The device can not.

`enumerator cnrtAttrSupportHostRegsiter`

1: The device supports host memory registration; 0: The device does not.

`enumerator cnrtAttrCacheSize`

Deprecated. The size of system cache in bytes.

`enumerator cnrtAttrMaxNum`

The last one.

3.5 enum cnrtDeviceP2PAttr_t

typedef enum {

cnrtDevP2PAttrAccessSupported = 0,

cnrtDevP2PAttrNativeAtomicSupported,

cnrtDevP2PAttrMaxNum,

} cnrtDeviceP2PAttr_t;

`enum cnrtDeviceP2PAttr_t`

Describes the P2P attributes of the MLU device.

Values:

`enumerator cnrtDevP2PAttrAccessSupported`

P2P access is enabled.

`enumerator cnrtDevP2PAttrNativeAtomicSupported`

Native atomic operation between the device and host is supported.

`enumerator cnrtDevP2PAttrMaxNum`

The last one.

3.6 enum cnrtDeviceLimit_t

typedef enum {

cnrtDevLimitStackSize = 0,

cnrtDevLimitPrintfFifoSize,

cnrtDevLimitMaxNum,

} cnrtDeviceLimit_t;

`enum cnrtDeviceLimit_t`

Deprecated. Describes the limits of the MLU device.

Values:

`enumerator cnrtDevLimitStackSize`

MLU stack size can be used for each MLU core.

`enumerator cnrtDevLimitPrintfFifoSize`

MLU print line FIFO size.

`enumerator cnrtDevLimitMaxNum`

The last one.

3.7 enum cnrtDeviceConfig_t

typedef enum {

cnrtDeviceConfigReserved = 0,

cnrtDeviceConfigPrintfFifoNum = 1,

cnrtDeviceConfigMaxPersistingL2CacheSize = 2,

cnrtDeviceConfigMaxNum,

} cnrtDeviceConfig_t;

`enum cnrtDeviceConfig_t`

Describes the device configurations.

Values:

`enumerator cnrtDeviceConfigReserved`

Reserved.

`enumerator cnrtDeviceConfigPrintfFifoNum`

The record number of print line FIFO per MLU core.

`enumerator cnrtDeviceConfigMaxPersistingL2CacheSize`

The maximum L2 persisting cache size in bytes.

`enumerator cnrtDeviceConfigMaxNum`

The maximum number of device configuration enums.

3.8 enum cnrtDeviceFlags_t

typedef enum {

cnrtDeviceScheduleSpin = 0,

cnrtDeviceScheduleBlock = 1,

cnrtDeviceScheduleYield = 2,

cnrtDeviceScheduleAuto = 3,

cnrtDeviceFlagsMaxNum,

} cnrtDeviceFlags_t;

`enum cnrtDeviceFlags_t`

Describes the device flags used for the current process execution on the current device.

The *cnrtGetDeviceFlag* API is used to retrieve the flags set.

Values:

`enumerator cnrtDeviceScheduleSpin`

CPU actively spins when waiting for the device execution result.

`enumerator cnrtDeviceScheduleBlock`

CPU thread is blocked on a synchronization primitive when waiting for the device execution results.

`enumerator cnrtDeviceScheduleYield`

CPU thread yields when waiting for the device execution results.

`enumerator cnrtDeviceScheduleAuto`

Automatic scheduling.

`enumerator cnrtDeviceFlagsMaxNum`

The last one.

3.9 enum cnrtMemTransDir_t

typedef enum {

cnrtMemcpyHostToDev = 0,

cnrtMemcpyDevToDev,

cnrtMemcpyDevToHost,

cnrtMemcpyHostToHost,

cnrtMemcpyPeerToPeer,

cnrtMemcpyNoDirection,

} cnrtMemTransDir_t;

`enum cnrtMemTransDir_t`

Describes the direction of data copying.

Values:

`enumerator cnrtMemcpyHostToDev`

Host to device.

`enumerator cnrtMemcpyDevToDev`

Data copy in a single device.

`enumerator cnrtMemcpyDevToHost`

Device to host.

`enumerator cnrtMemcpyHostToHost`

Host to host.

`enumerator cnrtMemcpyPeerToPeer`

P2P in two different devices.

`enumerator cnrtMemcpyNoDirection`

Data copying without a specified direction.

3.10 enum cnrtMemRangeAttribute_t

typedef enum {

cnrtMemRangeAttributePreferredLocation = 0,

cnrtMemRangeAttributeAccessedBy,

} cnrtMemRangeAttribute_t;

`enum cnrtMemRangeAttribute_t`

Describes the range attributes. Not used now.

Values:

`enumerator cnrtMemRangeAttributePreferredLocation`

The preferred location of the range.

`enumerator cnrtMemRangeAttributeAccessedBy`

The memory range set for the specified device.

3.11 enum cnrtMemType_t

typedef enum {

cnrtMemTypeUnregistered = 0,

cnrtMemTypeHost,

cnrtMemTypeDevice,

} cnrtMemType_t;

`enum cnrtMemType_t`

Describes memory types.

Values:

`enumerator cnrtMemTypeUnregistered`

Unregistered memory.

`enumerator cnrtMemTypeHost`

Host memory.

`enumerator cnrtMemTypeDevice`

Device memory.

3.12 enum cnrtUvaCacheMode_t

typedef enum {

cnrtUvaNotSupport = - 1,

cnrtUvaUnknown = 0,

cnrtUvaUnCached,

cnrtUvaCached,

} cnrtUvaCacheMode_t;

`enum cnrtUvaCacheMode_t`

Describes UVA cache modes.

Values:

`enumerator cnrtUvaNotSupport`

The current platform or driver version does not support this enum.

`enumerator cnrtUvaUnknown`

Unregistered cache mode.

`enumerator cnrtUvaUnCached`

Non-cacheable UVA.

`enumerator cnrtUvaCached`

Cacheable UVA.

3.13 enum cnrtHostAllocFlags_t

typedef enum {

cnrtHostAllocDefault = 0,

cnrtHostAllocMapped,

} cnrtHostAllocFlags_t;

`enum cnrtHostAllocFlags_t`

Describes properties of allocated memory. Not used now.

Values:

`enumerator cnrtHostAllocDefault`

Default host allocation type which is equal to host memory allocated by *cnrtHostMalloc*.

`enumerator cnrtHostAllocMapped`

Allocated host memory that is mapped to an MLU device.

3.14 enum cnrtDataType_t

typedef enum cnrtDataType {

cnrtInvalid = 0x0,

cnrtFloat16 = 0x12,

cnrtFloat32 = 0x13,

cnrtFloat64 = 0x14,

cnrtInt4 = 0x20,

cnrtInt8 = 0x21,

cnrtInt16 = 0x22,

cnrtInt32 = 0x23,

cnrtInt64 = 0x24,

cnrtAuto = 0x25,

cnrtUInt8 = 0x31,

cnrtUInt16 = 0x32,

cnrtUInt32 = 0x33,

cnrtFix8 = 0x41,

cnrtQuant8 = 0x51,

cnrtBool = 0x61,

} cnrtDataType_t;

`enum cnrtDataType`

Describes the data types supported by CNRT.

Values:

`enumerator cnrtInvalid`

Invalid data.

`enumerator cnrtFloat16`

16-bit floating-point data.

`enumerator cnrtFloat32`

32-bit floating-point data.

`enumerator cnrtFloat64`

64-bit floating-point data.

`enumerator cnrtInt4`

Not supported yet.

`enumerator cnrtInt8`

8-bit integer.

`enumerator cnrtInt16`

16-bit integer.

`enumerator cnrtInt32`

32-bit integer.

`enumerator cnrtInt64`

64-bit integer.

`enumerator cnrtAuto`

Automatic bit-width integer. It changes among int8, int16, etc.

`enumerator cnrtUInt8`

8-bit unsigned integer.

`enumerator cnrtUInt16`

16-bit unsigned integer.

`enumerator cnrtUInt32`

32-bit unsigned integer.

`enumerator cnrtFix8`

8-bit fixed-point data.

`enumerator cnrtQuant8`

8-bit data.

`enumerator cnrtBool`

Boolean type.

`typedef enum` *cnrtDataType* `cnrtDataType_t`

Describes the data types supported by CNRT.

3.15 enum cnrtDataType_V2_t

```
typedef enum cnrtDataType_V2 {
    cnrtUnknown = 0,
    cnrtDouble = 0x1,
    cnrtFloat = 0x2,
    cnrtHalf = 0x3,
    cnrtBfloat = 0x4,
    cnrtUlonglong = 0x11,
    cnrtUint = 0x12,
    cnrtUshort = 0x13,
    cnrtUchar = 0x14,
    cnrtLonglong = 0x21,
    cnrtInt = 0x22,
    cnrtShort = 0x23,
    cnrtChar = 0x24,
    cnrtBoolean = 0x31,
} cnrtDataType_V2_t;
```

`enum cnrtDataType_V2`

Describes the data types supported by CNRT.

Values:

enumerator cnrtUnknown

Invalid data.

enumerator cnrtDouble

64-bit floating-point data.

enumerator cnrtFloat

32-bit floating-point data.

enumerator cnrtHalf

16-bit floating-point data.

enumerator cnrtBfloat

BF16 data type.

enumerator cnrtUlonglong

64-bit unsigned integer.

enumerator cnrtUint

32-bit unsigned integer.

enumerator cnrtUshort

16-bit unsigned integer.

enumerator cnrtUchar

8-bit unsigned integer.

enumerator cnrtLonglong

64-bit integer.

enumerator cnrtInt

32-bit integer.

enumerator cnrtShort

16-bit integer.

enumerator cnrtChar

8-bit integer.

enumerator cnrtBoolean

Boolean type.

typedef enum *cnrtDataType_V2* cnrtDataType_V2_t

Describes the data types supported by CNRT.

3.16 enum cnrtRoundingMode_t

typedef enum cnrtRoundingMode {

cnrtRounding_rn = 0,

cnrtRounding_rz = 1,

cnrtRounding_rd = 2,

cnrtRounding_ru = 3,

cnrtRounding_ro = 4,

cnrtRounding_rm = 5,

cnrtRounding_max,

} cnrtRoundingMode_t;

`enum cnrtRoundingMode`

Describes the rounding mode supported by CNRT.

Values:

`enumerator cnrtRounding_rn`

Converts an input number in round-to-nearest-even mode.

`enumerator cnrtRounding_rz`

Converts an input number in round-to-zero mode.

`enumerator cnrtRounding_rd`

Converts an input number in round-down mode.

`enumerator cnrtRounding_ru`

Converts an input number in round-up mode.

`enumerator cnrtRounding_ro`

Converts an input number in round-off-zero mode.

`enumerator cnrtRounding_rm`

Converts an input number in round-to-math mode.

`enumerator cnrtRounding_max`

The last one.

`typedef enum` *cnrtRoundingMode* `cnrtRoundingMode_t`

Describes the rounding mode supported by CNRT.

3.17 enum cnrtNotifierFlags_t

typedef enum cnrtNotifierFlags {

CNRT_NOTIFIER_DEFAULT = 0x0,

CNRT_NOTIFIER_DISABLE_TIMING_SW = 0x2,

CNRT_NOTIFIER_DISABLE_TIMING = CNRT_NOTIFIER_DISABLE_TIMING_SW,

CNRT_NOTIFIER_DISABLE_TIMING_ALL = 0x4,

CNRT_NOTIFIER_INTERPROCESS = 0x8,

} cnrtNotifierFlags_t;

`enum cnrtNotifierFlags`

Describes the flags of notifier, which are used by *cnrtNotifierCreateWithFlags*.

Values:

`enumerator CNRT_NOTIFIER_DEFAULT`

The default notifier creation flag.

`enumerator CNRT_NOTIFIER_DISABLE_TIMING_SW`

The notifier will not record sw timestamp data.

`enumerator CNRT_NOTIFIER_DISABLE_TIMING`

Deprecated.

`enumerator CNRT_NOTIFIER_DISABLE_TIMING_ALL`

The notifier will not record timestamp data to reduce overhead.

`enumerator CNRT_NOTIFIER_INTERPROCESS`

The notifier is suitable for interprocess use. CNRT_NOTIFIER_DISABLE_TIMING_ALL must be set.

`typedef enum` *cnrtNotifierFlags* `cnrtNotifierFlags_t`

Describes the flags of notifier, which are used by *cnrtNotifierCreateWithFlags*.

3.18 enum cnrtCacheOps_t

typedef enum {

CNRT_FLUSH_CACHE = 1,

CNRT_INVALID_CACHE = 2,

} cnrtCacheOps_t;

`enum cnrtCacheOps_t`

Describes the cache operation types.

Values:

`enumerator CNRT_FLUSH_CACHE`

Flushes dcache of the host CPU.

`enumerator CNRT_INVALID_CACHE`

Invalidates dcache of the host CPU, which is currently reserved.

3.19 enum cnrtAccessProperty_t

typedef enum cnrtAccessProperty {

cnrtAccessPolicyNormal = 0,

cnrtAccessPolicyStreaming = 1,

cnrtAccessPolicyPersisting = 2,

} cnrtAccessProperty_t;

`enum cnrtAccessProperty`

Specifies performance hint for hitProp and missProp with *cnrtAccessPolicyWindow*.

Values:

`enumerator cnrtAccessPolicyNormal`

Normal cache persistance.

`enumerator cnrtAccessPolicyStreaming`

Streaming access is likely to persist in cache.

`enumerator cnrtAccessPolicyPersisting`

Persisting access is more likely to persist in cache.

`typedef enum` *cnrtAccessProperty* `cnrtAccessProperty_t`

Specifies performance hint for hitProp and missProp with *cnrtAccessPolicyWindow*.

3.20 enum cnrtQueueAttrID_t

typedef enum cnrtQueueAttrID {

cnrtQueueAttributeAccessPolicyWindow = 1,

} cnrtQueueAttrID_t;

`enum cnrtQueueAttrID`

Describes queue attributes.

Values:

`enumerator cnrtQueueAttributeAccessPolicyWindow`

Queue attribute ID used to change and query *cnrtAccessPolicyWindow*.

`typedef enum` *cnrtQueueAttrID* `cnrtQueueAttrID_t`

Describes queue attributes.

3.21 enum cnrtQueueCaptureStatus_t

typedef enum cnrtQueueCaptureStatus {

cnrtQueueCaptureStatusNone = 0,

cnrtQueueCaptureStatusActive = 1,

cnrtQueueCaptureStatusInvalidated = 2,

} cnrtQueueCaptureStatus_t;

`enum cnrtQueueCaptureStatus`

Queue capture statuses.

Values:

`enumerator cnrtQueueCaptureStatusNone`

A queue is not capturing.

`enumerator cnrtQueueCaptureStatusActive`

A queue is actively capturing.

`enumerator cnrtQueueCaptureStatusInvalidated`

A queue is partly capturing sequence that has been invalidated, but not terminated.

`typedef enum` *cnrtQueueCaptureStatus* `cnrtQueueCaptureStatus_t`

Queue capture statuses.

3.22 enum cnrtQueueCaptureMode_t

typedef enum cnrtQueueCaptureMode {

cnrtQueueCaptureModeGlobal = 0,

cnrtQueueCaptureModeThreadLocal = 1,

cnrtQueueCaptureModeRelaxed = 2,

} cnrtQueueCaptureMode_t;

`enum cnrtQueueCaptureMode`

Queue capture modes.

When a queue is capturing, it may affect potentially unsafe APIs.

The potentially unsafe APIs refer to memory allocation and queue synchronization related APIs, such as *cnrtQueueSync()*, *cnrtQueueQuery()* and *cnrtMalloc()*, etc, which may cause unexpected result per called when any queue is capturing.

cnrtQueueCaptureModeThreadLocal is not supported yet.

Values:

`enumerator cnrtQueueCaptureModeGlobal`

If any queue is actively capturing under *cnrtQueueCaptureModeGlobal* mode, all the potentially unsafe APIs are prohibited from calling.

`enumerator cnrtQueueCaptureModeThreadLocal`

If any queue is actively capturing under *cnrtQueueCaptureModeThreadLocal* mode, all the potentially unsafe APIs in local thread will be prohibited from calling.

`enumerator cnrtQueueCaptureModeRelaxed`

If there are only queue captures activated under *cnrtQueueCaptureModeRelaxed* mode, no potentially unsafe APIs will be prohibited from calling.

`typedef enum` *cnrtQueueCaptureMode* `cnrtQueueCaptureMode_t`

Queue capture modes.

When a queue is capturing, it may affect potentially unsafe APIs.

The potentially unsafe APIs refer to memory allocation and queue synchronization related APIs, such as *cnrtQueueSync()*, *cnrtQueueQuery()* and *cnrtMalloc()*, etc, which may cause unexpected result per called when any queue is capturing.

cnrtQueueCaptureModeThreadLocal is not supported yet.

3.23 enum cnrtUpdateQueueCaptureDependenciesFlags_t

typedef enum cnrtUpdateQueueCaptureDependenciesFlags {

cnrtQueueAddCaptureDependencies = 0,

cnrtQueueSetCaptureDependencies = 1,

} cnrtUpdateQueueCaptureDependenciesFlags_t;

`enum cnrtUpdateQueueCaptureDependenciesFlags`

Flags for *cnrtQueueUpdateCaptureDependencies*.

Values:

`enumerator cnrtQueueAddCaptureDependencies`

Adds new nodes to the dependency set.

`enumerator cnrtQueueSetCaptureDependencies`

Replaces dependency set with new nodes.

`typedef enum` *cnrtUpdateQueueCaptureDependenciesFlags* `cnrtUpdateQueueCaptureDependenciesFlags_t`

Flags for *cnrtQueueUpdateCaptureDependencies*.

3.24 enum cnrtUserObjectAcquireflags_t

typedef enum cnrtUserObjectAcquireflags {

cnrtTaskTopoUserObjectMove = 0x1,

} cnrtUserObjectAcquireflags_t;

`enum cnrtUserObjectAcquireflags`

Flags for acquiring user object references for Task Topo.

Values:

`enumerator cnrtTaskTopoUserObjectMove`

Transfers references from the caller rather than creating new references.

`typedef enum` *cnrtUserObjectAcquireflags* `cnrtUserObjectAcquireflags_t`

Flags for acquiring user object references for Task Topo.

3.25 enum cnrtTaskTopoNodeType_t

typedef enum cnrtTaskTopoNodeType {

cnrtTaskTopoNodeTypeEmpty = 0,

cnrtTaskTopoNodeTypeKernel = 1,

cnrtTaskTopoNodeTypeHost = 2,

cnrtTaskTopoNodeTypeMemcpy = 3,

cnrtTaskTopoNodeTypeMemset = 4,

cnrtTaskTopoNodeTypeTaskTopo = 5,

} cnrtTaskTopoNodeType_t;

`enum cnrtTaskTopoNodeType`

Task Topo node types.

Values:

`enumerator cnrtTaskTopoNodeTypeEmpty`

Empty node.

`enumerator cnrtTaskTopoNodeTypeKernel`

Kernel node.

`enumerator cnrtTaskTopoNodeTypeHost`

Host function node.

`enumerator cnrtTaskTopoNodeTypeMemcpy`

Memcpy node.

`enumerator cnrtTaskTopoNodeTypeMemset`

Memset node.

`enumerator cnrtTaskTopoNodeTypeTaskTopo`

Child Task Topo node.

`typedef enum` *cnrtTaskTopoNodeType* `cnrtTaskTopoNodeType_t`

Task Topo node types.

3.26 enum cnrtKernelNodeAttr_t

typedef enum cnrtKernelNodeAttr {

cnrtKernelNodeAttributeAccessPolicyWindow = 1,

} cnrtKernelNodeAttr_t;

`enum cnrtKernelNodeAttr`

Task Topo kernel node attributes.

Values:

`enumerator cnrtKernelNodeAttributeAccessPolicyWindow`

Identifier for *cnrtKernelNodeAttrValue::accessPolicyWindow*.

`typedef enum` *cnrtKernelNodeAttr* `cnrtKernelNodeAttr_t`

Task Topo kernel node attributes.

3.27 enum cnrtTaskTopoEntityUpdateResult_t

typedef enum cnrtTaskTopoEntityUpdateResult {

cnrtTaskTopoEntityUpdateSuccess = 0x0,

cnrtTaskTopoEntityUpdateError = 0x1,

cnrtTaskTopoEntityUpdateErrorTopologyChanged = 0x2,

cnrtTaskTopoEntityUpdateErrorNodeTypeChanged = 0x3,

cnrtTaskTopoEntityUpdateErrorParametersChanged = 0x4,

cnrtTaskTopoEntityUpdateErrorNotSupported = 0x5,

cnrtTaskTopoEntityUpdateErrorUnsupportedFunctionChange = 0x6,

cnrtTaskTopoEntityUpdateErrorAttributesChanged = 0x7,

} cnrtTaskTopoEntityUpdateResult_t;

`enum cnrtTaskTopoEntityUpdateResult`

Task Topo entity update error result.

Values:

`enumerator cnrtTaskTopoEntityUpdateSuccess`

The update succeeds.

`enumerator cnrtTaskTopoEntityUpdateError`

The update fails for an unexpected reason.

`enumerator cnrtTaskTopoEntityUpdateErrorTopologyChanged`

The update fails because the topology changed.

`enumerator cnrtTaskTopoEntityUpdateErrorNodeTypeChanged`

The update fails because a node type changed.

`enumerator cnrtTaskTopoEntityUpdateErrorParametersChanged`

The update fails because the parameters change in a way that is not supported.

`enumerator cnrtTaskTopoEntityUpdateErrorNotSupported`

The update fails because something about the node is not supported.

`enumerator cnrtTaskTopoEntityUpdateErrorUnsupportedFunctionChange`

The update fails because the function of a kernel node changed in an unsupported way.

`enumerator cnrtTaskTopoEntityUpdateErrorAttributesChanged`

The update fails because the node attributes changed in a way that is not supported.

`typedef enum` *cnrtTaskTopoEntityUpdateResult* `cnrtTaskTopoEntityUpdateResult_t`

Task Topo entity update error result.

3.28 enum cnrtTaskTopoDebugDotFlags_t

typedef enum cnrtTaskTopoDebugDotFlags {

cnrtTaskTopoDebugDotFlagsVerbose = (1 << 0),

cnrtTaskTopoDebugDotFlagsRuntimeTypes = (1 << 1),

cnrtTaskTopoDebugDotFlagsHandles = (1 << 2),

cnrtTaskTopoDebugDotFlagsKernelNodeParams = (1 << 3),

cnrtTaskTopoDebugDotFlagsMemcpyNodeParams = (1 << 4),

cnrtTaskTopoDebugDotFlagsMemsetNodeParams = (1 << 5),

cnrtTaskTopoDebugDotFlagsHostNodeParams = (1 << 6),

cnrtTaskTopoDebugDotFlagsKernelNodeAttribute = (1 << 7),

} cnrtTaskTopoDebugDotFlags_t;

`enum cnrtTaskTopoDebugDotFlags`

The additional write flags to create DOT file.

Values:

`enumerator cnrtTaskTopoDebugDotFlagsVerbose`

Outputs all debug data as if every debug flag is enabled.

`enumerator cnrtTaskTopoDebugDotFlagsRuntimeTypes`

Uses runtime structs for output.

`enumerator cnrtTaskTopoDebugDotFlagsHandles`

Adds handles to output.

`enumerator cnrtTaskTopoDebugDotFlagsKernelNodeParams`

Adds *cnrtKernelNodeParams_t* values to output.

`enumerator cnrtTaskTopoDebugDotFlagsMemcpyNodeParams`

Adds *cnrtMemcpy3dParam_t* values to output. .

`enumerator cnrtTaskTopoDebugDotFlagsMemsetNodeParams`

Adds *cnrtMemsetParams_t* values to output .

`enumerator cnrtTaskTopoDebugDotFlagsHostNodeParams`

Adds *cnrtHostNodeParams_t* values to output.

`enumerator cnrtTaskTopoDebugDotFlagsKernelNodeAttribute`

Adds *cnrtKernelNodeAttrValue_t* values to output .

`typedef enum` *cnrtTaskTopoDebugDotFlags* `cnrtTaskTopoDebugDotFlags_t`

The additional write flags to create DOT file.

3.29 union cnrtQueueAttrValue_t

typedef union cnrtQueueAttrValue {

cnrtAccessPolicyWindow_t accessPolicyWindow;

} cnrtQueueAttrValue_t;

`union cnrtQueueAttrValue`

#include <cnrt.h> Describes the queue attribute union, which is used with *cnrtQueueGetAttribute()* and *cnrtQueueSetAttribute()*.

Public Members

cnrtAccessPolicyWindow_t `accessPolicyWindow`

Queue attribute value for *cnrtAccessPolicyWindow*.

`typedef union` *cnrtQueueAttrValue* `cnrtQueueAttrValue_t`

Describes the queue attribute union, which is used with *cnrtQueueGetAttribute()* and *cnrtQueueSetAttribute()*.

3.30 union cnrtKernelNodeAttrValue_t

typedef union cnrtKernelNodeAttrValue {

cnrtAccessPolicyWindow_t accessPolicyWindow;

} cnrtKernelNodeAttrValue_t;

`union cnrtKernelNodeAttrValue`

#include <cnrt.h>

Task Topo kernel node attribute value union, which is used by *cnrtTaskTopoKernelNodeSetAttribute* and *cnrtTaskTopoKernelNodeGetAttribute*.

Public Members

cnrtAccessPolicyWindow_t `accessPolicyWindow`

Kernel node attribute value for *cnrtAccessPolicyWindow_t*.

`typedef union` *cnrtKernelNodeAttrValue* `cnrtKernelNodeAttrValue_t`

Task Topo kernel node attribute value union, which is used by *cnrtTaskTopoKernelNodeSetAttribute* and *cnrtTaskTopoKernelNodeGetAttribute*.

3.31 struct cnrtPointerAttributes_t

typedef struct {

cnrtMemType_t type;

int device;

size_t size;

void * devicePointer;

void * hostPointer;

cnrtUvaCacheMode_t cacheMode;

void * deviceBasePointer;

} cnrtPointerAttributes_t;

`struct cnrtPointerAttributes_t`

Describes the pointer attributes.

Public Members

cnrtMemType_t `type`

The memory type.

int `device`

Device ordinal which the pointer is allocated from.

size_t `size`

Size in bytes of the pointer from *cnrtMalloc*.

void *`devicePointer`

Device pointer related to the pointer.

void *`hostPointer`

Host pointer related to the pointer.

cnrtUvaCacheMode_t `cacheMode`

Cache mode of the host pointer.

void *`deviceBasePointer`

The base address of the device pointer.

3.32 struct cnrtUUID_st

struct cnrtUUID_st {

char uuid[16];

};

`struct cnrtUUID_st`

3.33 struct cnrtDeviceProp_t

```
typedef struct cnrtDeviceProp_V3 {
    char name[256];
    int totalMem;
    int maxDim[3];
    int ipuClockRate;
    int memClockRate;
    int totalConstMem;
    int major;
    int minor;
    int ECCEnabled;
    int pciBusID;
    int pciDeviceID;
    int pciDomainID;
    int maxL2CacheSize;
    int persistingL2CacheMaxSize;
    int queuePrioritiesSupported;
    int sparseComputingSupported;
    int FP16ComputingSupported;
    int INT4ComputingSupported;
    int INT8ComputingSupported;
    int BF16ComputingSupported;
    int TF32ComputingSupported;
    int maxQueueSize;
    int maxNotifierSize;
    int tinyCoreSupported;
    int codecJPEGSupported;
    int codecH264Supported;
    int codecH265Supported;
    int maxClusterCountPerUnionTask;
    int clusterCount;
    int McorePerCluster;
    int maxQuadrantCount;
    int maxUnionTypePerQuadrant;
    int maxClusterPerUnionLimitTask;
    int ISAVersion;
    int isMultipleTensorProcessor;
    int NramSizePerMcore;
    int WramSizePerMcore;
    int LmemSizePerMcore;
    int SramSizePerMcore;
    int globalMemoryNodeCount;
    int cacheSize;
    int GmemBusWidth;
    int computeMode;
    int MDRMemorySize;
    int availableGlobalMemorySize;
    cnrtUUID_t uuid;
    int reserved[32];
} cnrtDeviceProp_t;
```

`struct cnrtDeviceProp_V3`

Describes the properties of the MLU device.

Public Members

char `name`[256]

MLU device name.

int `totalMem`

Total memory available on device in MB.

int `maxDim`[3]

The maximum size of each dimension of a block.

int `ipuClockRate`

Cluster clock frequency in kilohertz.

int `memClockRate`

Memory clock frequency in kilohertz.

int `totalConstMem`

Memory available on device for **mlu_const** variables in MB.

int `major`

Major compute capability of the MLU device.

int `minor`

Minor compute capability of the MLU device.

int `ECCEnabled`

The device has ECC support enabled.

int `pciBusID`

PCI bus identifier of the MLU device.

int `pciDeviceID`

PCI device identifier of the MLU device.

int `pciDomainID`

PCI domain ID of the MLU device.

int `maxL2CacheSize`

The size of L2 cache in bytes.

int `persistingL2CacheMaxSize`

The maximum L2 persisting cache size in bytes.

int `queuePrioritiesSupported`

1: The device supports setting queue priorities; 0: The device does not.

int `sparseComputingSupported`

1: The device supports sparse computing; 0: The device does not.

int `FP16ComputingSupported`

1: The device supports FP16; 0: The device does not.

int `INT4ComputingSupported`

1: The device supports INT4; 0: The device does not.

int `INT8ComputingSupported`

1: The device supports INT8; 0: The device does not.

int `BF16ComputingSupported`

1: The device supports BF16; 0: The device does not.

int `TF32ComputingSupported`

1: The device supports TF32; 0: The device does not.

int `maxQueueSize`

The maximum number of queues.

int `maxNotifierSize`

The maximum number of Notifiers.

int `tinyCoreSupported`

1: The device supports using tiny core to accelerate collective inter-device or intra-device communication; 0: The device does not.

int `codecJPEGSupported`

1: The device supports hardware JPEG codec acceleration; 0: The device does not.

int `codecH264Supported`

1: The device supports hardware video H.264 codec acceleration; 0: The device does not.

int `codecH265Supported`

1: The device supports hardware video H.265 codec acceleration; 0: The device does not.

int `maxClusterCountPerUnionTask`

The maximum number of clusters per union task.

int `clusterCount`

The maximum number of clusters of the MLU device.

int `McorePerCluster`

The maximum number of MLU Cores of each cluster.

int `maxQuadrantCount`

The maximum count of quadrants per device. Intra-quadrant clusters have the best unified memory access performance.

int `maxUnionTypePerQuadrant`

The maximum union task types that can maintain unified intra-quadrant memory access.

int `maxClusterPerUnionLimitTask`

The maximum number of clusters per union limitation task.

int `ISAVersion`

ISA version of current MLU device in the form of three-digit number.

int `isMultipleTensorProcessor`

1: The device adopts multi-tensor-processor architecture; 0: The device does not.

int `NramSizePerMcore`

The maximum nram memory available of each MLU Core in bytes.

int `WramSizePerMcore`

The maximum wram memory available of each MLU Core in bytes.

int `LmemSizePerMcore`

The maximum local memory available of each Core in MB.

int `SramSizePerMcore`

The maximum sram memory available of each MLU Core in bytes.

int `globalMemoryNodeCount`

The number of NUMA nodes on device.

int `cacheSize`

The size of system cache in bytes.

int `GmemBusWidth`

Global memory bus width in bits.

int `computeMode`

The compute mode that the device is currently in. See *cnrtComputeMode_t* for details.

int `MDRMemorySize`

MDR memory size in megabytes.

int `availableGlobalMemorySize`

Total available global memory size in MB.

cnrtUUID_t `uuid`

Universally Unique Identifier of the device.

int `reserved`[32]

reserved.

`typedef struct` *cnrtDeviceProp_V3* `cnrtDeviceProp_t`

Describes the properties of the MLU device.

Parameter for API call.

3.34 struct cnrtDim3_t

typedef struct {

unsigned int x;

unsigned int y;

unsigned int z;

} cnrtDim3_t;

`struct cnrtDim3_t`

Describes grid dimensions used for task execution.

Public Members

unsigned int `x`

The X axis. The value of X equals to: the number of tasks to run on each core multiplies 4.

unsigned int `y`

The Y axis. Each task is to run `y*z` times.

unsigned int `z`

The Z axis. Each task is to run `y*z` times.

3.35 struct cnrtPos_t

typedef struct {

size_t x;

size_t y;

size_t z;

} cnrtPos_t;

`struct cnrtPos_t`

The offset of the address.

Public Members

size_t `x`

The offset in the X direction.

size_t `y`

The offset in the Y direction.

size_t `z`

The offset in the Z direction.

3.36 struct cnrtPitchedPtr_t

typedef struct {

size_t pitch;

void * ptr;

size_t xsize;

size_t ysize;

} cnrtPitchedPtr_t;

`struct cnrtPitchedPtr_t`

The pitch (alignment) of the address. None of the parameters can be 0.

Public Members

size_t `pitch`

The pitch of the memory. It cannot be less than the p->extent.width, or greater than 4MB.

void *`ptr`

The pointer of the memory. The same as the p->dst.

size_t `xsize`

The memory X size (reserved). It is set to p->extent.width.

size_t `ysize`

The memory Y size. It cannot be less than the p->extent.height, or greater than 4MB.

3.37 struct cnrtExtent_t

typedef struct {

size_t depth;

size_t height;

size_t width;

} cnrtExtent_t;

`struct cnrtExtent_t`

The extent (size) of the address. None of the parameters can be 0.

Public Members

size_t `depth`

The depth of the memory.

size_t `height`

The height of the memory. It cannot be greater than 1MB.

size_t `width`

The width of the memory. It cannot be greater than 1MB.

3.38 struct cnrtMemcpy3dParam_t

typedef struct cnrtMemcpy3dParam_st {

void * dst;

cnrtPos_t dstPos;

cnrtPitchedPtr_t dstPtr;

cnrtExtent_t extent;

cnrtMemTransDir_t dir;

void * src;

cnrtPos_t srcPos;

cnrtPitchedPtr_t srcPtr;

} cnrtMemcpy3dParam_t;

`struct cnrtMemcpy3dParam_st`

The configuration parameters of 3D memory copy.

Public Members

void *`dst`

The destination address.

cnrtPos_t `dstPos`

The destination address position.

cnrtPitchedPtr_t `dstPtr`

The pitch of the destination address.

cnrtExtent_t `extent`

The extent of the memory.

cnrtMemTransDir_t `dir`

Data copy direction.

void *`src`

The source address.

cnrtPos_t `srcPos`

The source address position.

cnrtPitchedPtr_t `srcPtr`

The pitch of the source address.

`typedef struct` *cnrtMemcpy3dParam_st* `cnrtMemcpy3dParam_t`

The configuration parameters of 3D memory copy.

3.39 struct cnrtIpcNotifierHandle_v1

typedef struct cnrtIpcNotifierHandle_st {

uint64_t res[CNRT_IPC_HANDLE_SIZE];

} cnrtIpcNotifierHandle_v1;

`struct cnrtIpcNotifierHandle_st`

The IPC notifier handle.

Public Members

uint64_t `res`[8]

Reserved for IPC notifier handle.

`typedef struct` *cnrtIpcNotifierHandle_st* `cnrtIpcNotifierHandle_v1`

The IPC notifier handle.

3.40 struct cnrtAccessPolicyWindow_t

typedef struct cnrtAccessPolicyWindow {

void * baseAddr;

size_t numBytes;

float hitRatio;

enum cnrtAccessProperty{

} hitProp;

enum cnrtAccessProperty{

} missProp;

} cnrtAccessPolicyWindow_t;

`struct cnrtAccessPolicyWindow`

Public Members

void *`baseAddr`

Starting address of access policy window.

size_t `numBytes`

Size in bytes of access policy window. MLU driver may restrict the maximum size and alignment.

float `hitRatio`

hitRatio specifies percentage of cache lines assigned hitProp, the rest are assigned missProp. Valid range is [0, 1.0].

`enum` *cnrtAccessProperty* `hitProp`

The access property set for cache hit in *cnrtAccessProperty*.

enum *cnrtAccessProperty* `missProp`

The access property set for cache miss in *cnrtAccessProperty*. It must be either *cnrtAccessPolicyNormal* or *cnrtAccessPolicyStreaming*.

`typedef struct` *cnrtAccessPolicyWindow* `cnrtAccessPolicyWindow_t`

3.41 struct cnrtHostNodeParams_t

typedef struct cnrtHostNodeParams_st {

cnrtHostFn_t fn;

void * userData;

} cnrtHostNodeParams_t;

`struct cnrtHostNodeParams_st`

Host node parameters.

Public Members

cnrtHostFn_t `fn`

The API to call when the node is being executed.

void *`userData`

The argument to be passed to the API.

`typedef struct` *cnrtHostNodeParams_st* `cnrtHostNodeParams_t`

Host node parameters.

3.42 struct cnrtKernelNodeParams_t

typedef struct cnrtKernelNodeParams_st {

void * func;

cnrtDim3_t dim;

cnrtFunctionType_t type;

unsigned int reserve;

void * * kernelParams;

void * * extra;

} cnrtKernelNodeParams_t;

`struct cnrtKernelNodeParams_st`

Kernel node parameters.

Public Members

void *`func`

The kernel to invoke.

cnrtDim3_t `dim`

The grid dimensions.

cnrtFunctionType_t `type`

The union type of kernel.

unsigned int `reserve`

Reserved parameter.

void **`kernelParams`

The array of pointers to kernel parameters.

void **`extra`

Extra options, such as packaged parameters.

`typedef struct` *cnrtKernelNodeParams_st* `cnrtKernelNodeParams_t`

Kernel node parameters.

3.43 struct cnrtMemsetParams_t

typedef struct cnrtMemsetParams {

void * dst;

size_t pitch;

unsigned int value;

unsigned int elementSize;

size_t width;

size_t height;

} cnrtMemsetParams_t;

`struct cnrtMemsetParams`

Memset node parameters.

Public Members

void *`dst`

Destination device pointer.

size_t `pitch`

Pitch of destination device pointer, which will be ignored if the height is 1.

unsigned int `value`

Value to be set.

unsigned int `elementSize`

The size of each element in bytes, which must be 1, 2, or 4.

size_t `width`

Width of the row in elements.

size_t `height`

Number of rows.

`typedef struct` *cnrtMemsetParams* `cnrtMemsetParams_t`

Memset node parameters.

3.44 typedef cnrtJobType_t

`typedef` *cnrtFunctionType_t* `cnrtJobType_t`

3.45 typedef cnrtUUID_t

`typedef struct` *cnrtUUID_st* `cnrtUUID_t`

UUID(Universally Unique Identifier) of the device.

3.46 typedef cnrtQuantizedParam_t

`typedef struct` cnrtQuantizedParam *`cnrtQuantizedParam_t`

A pointer to the struct of the parameters that are quantized.

3.47 typedef cnrtIpcNotifierHandle

typedef *cnrtIpcNotifierHandle_v1* cnrtIpcNotifierHandle

3.48 typedef cnrtIpcMemHandle

typedef void *cnrtIpcMemHandle

The IPC memory handle. Pointer to void by default.

3.49 typedef cnrtQueue_t

typedef struct CNqueue_st *cnrtQueue_t

A pointer to the cnrtQueue struct holding the information about a queue.

The *cnrtQueueCreate* and *cnrtQueueDestroy* APIs are used to create and destroy an instance of cnrtQueue_t respectively.

Note

- This struct CNqueue_st is the same as struct cnrtQueue, which is defined in the versions before v5.1.1. After version v5.1.2, CNqueue_st is used to avoid compile warning when called mixedly with CNDrv.
- A definition from cnrtQueue to CNqueue_st is used to keep compatibility.

3.50 typedef cnrtNotifier_t

typedef struct CNnotifier_st *cnrtNotifier_t

A pointer to the cnrtNotifier struct holding the information about a notifier.

The *cnrtNotifierCreate* and *cnrtNotifierDestroy* APIs are used to create and destroy an instance of cnrtNotifier_t respectively.

Note

- This struct CNnotifier_st is the same as struct cnrtNotifier, which is defined in the versions before v5.1.1. After version v5.1.2, CNnotifier_st is used to avoid compiling warning when called mixedly with CNDrv.
- A definition from cnrtNotifier to CNnotifier_st is used to keep compatibility.

3.51 typedef cnrtTaskTopo_t

`typedef struct CNtaskTopo_st *cnrtTaskTopo_t`

Describes a pointer to the *cnrtTaskTopo_t* struct holding the information about a Task Topo.

3.52 typedef cnrtTaskTopoNode_t

`typedef struct CNtaskTopoNode_st *cnrtTaskTopoNode_t`

A pointer to the *cnrtTaskTopoNode_t* struct holding the information about a node of a Task Topo.

3.53 typedef cnrtHostFn_t

```
typedef void (*cnrtHostFn_t)(void *usreData)
```

CNRT host function.

3.54 typedef cnrtUserObject_t

`typedef struct CNuserObject_st *cnrtUserObject_t`

A pointer to the *cnrtUserObject_t* struct holding the information about a user object.

3.55 typedef cnrtTaskTopoEntity_t

`typedef struct CNtaskTopoEntity_st *cnrtTaskTopoEntity_t`

Executable Task Topo entity.

4 API Reference

4.1 Device Management

4.1.1 cnrtDeviceGetAttribute

cnrtRet_t **cnrtDeviceGetAttribute**(int **pValue*, *cnrtDeviceAttr_t* *attr*, int *device*)

Retrieves the information about the device.

Retrieves the `device` attributes of `attr` in `pValue`. See the supported `attr` in *cnrtDeviceAttr_t*.

Parameters

- `[out] pValue`: Pointer to device attribute value.
- `[in] attr`: The device attribute to retrieve.
- `[in] device`: The device ordinal to retrieve.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoDevice*

Note

- None.

Example

```
int main () {
  int ordinal = -1;
  cnrtGetDevice(&ordinal);
  int value = 0;
  cnrtDeviceGetAttribute(&value, cnrtAttrClusterCount, ordinal);
  printf("device: %d, cnrtAttrClusterCount: %d.\n", ordinal, value);

  return 0;
}
```

4.1.2 cnrtGetDeviceProperties

cnrtRet_t `cnrtGetDeviceProperties`(*cnrtDeviceProp_t* **prop*, int *device*)

Retrieves the information about the MLU device.

Retrieves the `prop` properties of ordinal `device`. See details in *cnrtDeviceProp_t*.

Parameters

- `[out] prop`: Pointer to the returned struct *cnrtDeviceProp_t*.
- `[in] device`: Device ordinal to retrieve.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoDevice*

Note

- None.

Example

```
int main () {
  int ordinal = -1;
  cnrtGetDevice(&ordinal);
  cnrtDeviceProp_t prop;
  cnrtGetDeviceProperties(&prop, ordinal);
  printf("device: %d, device name: %s.\n", ordinal, prop.name);

  return 0;
}
```

4.1.3 cnrtGetDevice

cnrtRet_t `cnrtGetDevice`(int **pOrdinal*)

Retrieves which device is currently being used.

Retrieves the current MLU device ordinal being used in `pOrdinal`.

Parameters

- `[out] pOrdinal`: Pointer to the device ordinal being used.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoDevice*

Note

- None.

Example

- See example in *cnrtGetDeviceProperties*.

4.1.4 cnrtSetDevice

cnrtRet_t `cnrtSetDevice`(int *ordinal*)

Sets the device to be used.

Sets the currently used MLU device to `ordinal` for the calling host thread.

Parameters

- `[in] ordinal`: The device ordinal to be used.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoDevice*

Note

- The device `ordinal` ranges in [0, *cnrtGetDeviceCount()* - 1].
- All the device memory allocated by *cnrtMalloc* after the API is called will be physically on the device `ordinal`. All the host memory allocated by *cnrtHostMalloc* will be associated with device `ordinal`. All the queues and notifiers will be associated with device `ordinal`. All the kernel launches will be executed on the device `ordinal`.

Example

- See example in *cnrtMemcpyPeer*.

4.1.5 cnrtGetDeviceCount

cnrtRet_t `cnrtGetDeviceCount`(unsigned int **pCount*)

Retrieves the number of MLU devices.

Retrieves in `pCount` the number of MLU devices in current system.

Parameters

- `[out] pCount`: Pointer to the number of MLU devices.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- See example in *cnrtMemcpyPeer*.

4.1.6 cnrtDeviceReset

cnrtRet_t `cnrtDeviceReset`(void)

Destroys all device resources and resets all state on the current device in the current process.

Releases all the allocated resources on the current device in the current process. If *cnrtDeviceReset* is called, any subsequent API call to this device will reinitialize the current device.

Return

- *cnrtSuccess*, *cnrtErrorNoDevice*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- Once this API is called, the device will be initialized immediately. So synchronize the host threads, otherwise coredump may occur.

Example

```
int main () {
  size_t size = sizeof(size_t) * N;
  void *d = NULL;
  cnrtMalloc((void **)&d, size);
  cnrtDeviceReset();

  if (cnrtSuccess == cnrtFree(d)) {
    printf("device reset failed.\n");
    return -1;
  }

  return 0;
}
```

4.1.7 cnrtSyncDevice

cnrtRet_t `cnrtSyncDevice`(void)

Waits for the current device in the current process to complete all the preceding tasks.

Blocks any further executions on the host until all the operations in the queues in the current process on the current MLU device are completed absolutely.

Return

- *cnrtSuccess*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtGetLastError*.

4.1.8 cnrtGetDeviceFlag

cnrtRet_t `cnrtGetDeviceFlag`(uint32_t **pFlags*)

Retrieves the device flags used for the current process executions on the current device. See *cnrtSetDeviceFlag()* for flag values.

Returns in `pFlags` how CPU interacts with the OS scheduler when waiting for the device execution result.

Parameters

- `[out] pFlags`: Pointer to the flags specifies whether the CPU thread yields or spins when waiting for the device execution result.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoDevice*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtSetDeviceFlag*.

4.1.9 cnrtSetDeviceFlag

cnrtRet_t `cnrtSetDeviceFlag`(uint32_t *flags*)

Sets flags used for the current process executions on the current device.

After initializing the current device, this API specifies whether the CPU thread yields or spins when waiting for the device execution result in the `flags`. The flags of input parameters determine the CPU thread scheduler patterns, which include:

- cnrtDeviceScheduleSpin: Allows to spin actively when waiting for results from device. This pattern decreases latency but may lower the performance of CPU threads and high CPU usage.
- cnrtDeviceScheduleBlock: The thread is blocked with low CPU usage, and enters sleep mode.
- cnrtDeviceScheduleYield: Allows to yield current thread when waiting for results from device. This pattern may increase latency but increase the performance of CPU threads.
- cnrtDeviceScheduleAuto: Automatic scheduling. According to the device number and the number of processors in the system, choose the mode cnrtDeviceScheduleSpin or cnrtDe-

viceScheduleBlock.

Parameters

- [in] flags: Options of how CPU interacts with the OS scheduler when waiting for the device execution result. See supported flags in *cnrtDeviceFlags_t*.

Return

- *cnrtSuccess*, *cnrtErrorNoDevice*, *cnrtErrorCndrvFuncCall*, *cnrtErrorSetOnActiveProcess*

Note

- This API should be called when Shared Context is inactive, and before calling *cnrtMalloc*, *cnrtQueueCreate*, etc. For more details about Shared Context, see "Cambricon CNRT Upgrade Guide".

Example

```
int main () {
  unsigned int flag = 0xFFFFFFFF;
  cnrtGetDeviceFlag(&flag);
  printf("default device flag: %d.\n", flag);

  unsigned int setFlag = (unsigned int)cnrtDeviceScheduleBlock;
  cnrtSetDeviceFlag(setFlag);
  cnrtGetDeviceFlag(&flag);
  printf("device flag after set flag: %d.\n", flag);

  return 0;
}
```

4.1.10 cnrtDeviceGetConfig

cnrtRet_t cnrtDeviceGetConfig(int64_t **pValue*, *cnrtDeviceConfig_t* *type*)

Gets device resource configurations.

Returns in pValue the current value of the configuration type.

Parameters

- [out] pValue: Returned value of configuration type.
- [in] type: Configuration type.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

```
int main() {
  int64_t value;
  cnrtRet_t res = cnrtDeviceSetConfig(cnrtDeviceConfigPrintfFifoNum, 4096);
  assert(res == cnrtSuccess);
  res = cnrtDeviceGetConfig(&value, cnrtDeviceConfigPrintfFifoNum);
  assert(res == cnrtSuccess && value == 4096);
}
```

4.1.11 cnrtDeviceSetConfig

cnrtRet_t `cnrtDeviceSetConfig`(*cnrtDeviceConfig_t type*,
int64_t *value*)

Sets device resource configurations.

Sets `type` to `value` to update the current configuration maintained by the device. The `value` may be rounded up or down to the nearest legal value.

Parameters

- `[in] type`: The configuration type to set.
- `[in] value`: The value of configuration type.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtDeviceGetConfig()*.

4.1.12 cnrtDeviceQueryKernelMemoryUsage

cnrtRet_t `cnrtDeviceQueryKernelMemoryUsage`(size_t **pBytes*)

Gets the device memory usage of all kernels on current device.

Parameters

- `[out] pBytes`: The size of the device memory used by all kernels.

Return

- *cnrtSuccess*, *cnrtErrorNoDevice*, *cnrtErrorSysNoMem*

Note

- This API does not trigger kernels loaded to the device.

Example

```
int main() {
  size_t kernel_memory_usage;
  cnrtSetDevice(0);
  cnrtDeviceQueryKernelMemoryUsage(&kernel_memory_usage);
}
```

4.1.13 cnrtAcquireMemHandle

cnrtRet_t `cnrtAcquireMemHandle`(*cnrtIpcMemHandle* **handle*,
void **ptr*)

Retrieves the inter-process memory handle for an allocated host or MLU device memory within the same process.

Returns the inter-process memory handle `handle` with the given host or MLU device memory pointed by `ptr` within the same process. The host or MLU device memory must be allocated by the *cnrtHostMalloc* or *cnrtMalloc*.

Parameters

- `[out] handle`: Pointer to the unique inter-process handle for the host or MLU device memory to share.
- `[in] ptr`: The base pointer to the allocated host or MLU device memory.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

```
int main () {
  int fd[2];
  if (pipe(fd) < 0) {
    printf("pipe fd failed.\n");
    return -1;
  }

  size_t size = 0x10000000;
  void *d_a = NULL;
  cnrtIpcMemHandle handle;
  pid_t pid = fork();
```

```
if (pid == 0) {
  close(fd[1]);
  void *address = NULL;
  if (!read(fd[0], &handle, sizeof(handle))) {
    printf("read err.\n");
    return;
  }

  // Map the inter-process memory handle exported from another process
  cnrtMapMemHandle((void **)&address, handle, 0);
  cnrtMemset(address, 2, size);
  cnrtUnMapMemHandle(address);
  printf("child process dev addr[%p] mem handle[%p]\n", address, handle);
  exit(EXIT_SUCCESS);
} else {
  cnrtMalloc((void **)&d_a, size);
  cnrtMemset(d_a, 1, size);

  // Acquire an inter-process memory handle
  cnrtAcquireMemHandle(&handle, d_a);
  printf("parent process d_a[%p] mem handle[%p]\n", d_a, handle);
  close(fd[0]);
  if (!write(fd[1], &handle, sizeof(handle))) {
    printf("write err.\n");
    return;
  }
  int status = -1;
  if (waitpid(pid, &status, 0) < 0) {
    printf("%s, waitpid error.\n", __func__);
    exit(EXIT_FAILURE);
  }
  EXPECT_EQ(WEXITSTATUS(status), EXIT_SUCCESS);
  EXPECT_NE(WIFEXITED(status), 0);
  char *h_a = (char *)malloc(size);
  cnrtMemcpy((void *)h_a, (void *)d_a, size, cnrtMemcpyNoDirection);
  for (size_t i = 0; i < size; i++) {
    if (2 != h_a[i]) {
      printf("data copy error.\n");
      break;
    }
  }
}
```

```
  return 0;
}
```

4.1.14 cnrtMapMemHandle

cnrtRet_t **cnrtMapMemHandle**(void ***ptr*,
                cnrtIpcMemHandle *handle*,
                int *flags*)

Maps an inter-process memory handle exported from another process and returns a device pointer usable in the local process.

Maps an inter-process memory handle `handle` shared by another process into the memory address space of the current MLU device or host. Returns the host or MLU device memory pointer pointed by `memPtr` used in the local process.

Parameters

- `[out] ptr`: Pointer to the host or MLU device memory.
- `[in] handle`: The inter-process memory handle shared by another process.
- `[in] flags`: Flags used in this operation. Currently it only supports value 0.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtAcquireMemHandle*.

4.1.15 cnrtUnMapMemHandle

cnrtRet_t **cnrtUnMapMemHandle**(void **ptr*)

Attempts to close memory mapped with *cnrtMapMemHandle*.

Unmaps the mapping between the memory address space of the current device or host pointed by `ptr` and the inter-process memory handle shared by another process. The mapping relation is created by *cnrtMapMemHandle*.

Parameters

- `[in] ptr`: Pointer to the host or MLU device memory.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtAcquireMemHandle*.

4.1.16 cnrtGetPeerAccessibility

```
cnrtRet_t cnrtGetPeerAccessibility(unsigned int *canAccess,
                                   int ordinal,
                                   int peerOrdinal)
```

Queries if an MLU device is capable of directly accessing data on another MLU device.

Returns in `canAccess` if an MLU device `ordinal` can access data on another MLU device `peerOrdinal`. If `ordinal` can access data on another MLU device `peerOrdinal`, *canAccess is 1, otherwise it is 0.

Parameters

- `[out] canAccess`: Pointer to the return value 1 or 0. If `ordinal` is able to access memories on another MLU device `peerOrdinal`, the return value is 1, otherwise the return value is 0.
- `[in] ordinal`: The ordinal of the device if it can directly access memories on another device. Call *cnrtGetDeviceCount* to get the total number of devices in the system. The device ID is in the range [0, *cnrtGetDeviceCount()* –1].
- `[in] peerOrdinal`: The ordinal of the device on which memories can be directly accessed. Call *cnrtGetDeviceCount* to get the total number of devices in the system. The ID of the device is in the range [0, *cnrtGetDeviceCount()* –1].

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorDeviceInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtMemcpyPeer*.

4.1.17 cnrtDeviceGetPCIBusId

cnrtRet_t `cnrtDeviceGetPCIBusId`(char **pciBusId*, int *len*, int *device*)

Retrieves the PCI bus ID for the required device.

Returns a string `pciBusId` of the `device` with a specified `len`.

Parameters

- `[out] pciBusId`: Pointer to the identifier for the device with the format [domain]:[bus]:[device].[function]. The `domain`, `bus`, `device`, `function` will be shown as hexadecimal values.
- `[in] len`: The specified maximum length of string.
- `[in] device`: The device ordinal.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorDeviceInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtDeviceGetByPCIBusId* API.

4.1.18 cnrtDeviceGetByPCIBusId

cnrtRet_t `cnrtDeviceGetByPCIBusId`(int **device*, `const` char **pciBusId*)

Retrieves a pointer to device.

Returns in the `device` by giving a PCI bus ID `pciBusId`.

Parameters

- `[out] device`: Pointer to the device ordinal.
- `[in] pciBusId`: The string in one of the following forms: [domain]:[bus]:[device].[function][domain]:[bus]:[device][bus]:[device].[function] where `domain`, `bus`, `device` and `function` are all hexadecimal values.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorDeviceInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

```
int main () {
  unsigned int count = 0;
  cnrtGetDeviceCount(&count);
  char str[100];
  int device = -1, ordinal = -1;
  for (unsigned int i = 0; i < count; i++) {
    cnrtSetDevice(i);
    cnrtDeviceGetPCIBusId(str, 100, i);
    cnrtDeviceGetByPCIBusId(&device, str);
    cnrtGetDevice(&ordinal);
    if (device != ordinal) {
      printf("Error: not the same device ordinal.\n");
    }
  }
  return 0;
}
```

4.1.19 cnrtDeviceGetQueuePriorityRange

```
cnrtRet_t cnrtDeviceGetQueuePriorityRange(int *pMinPriority,
                                          int *pMaxPriority)
```

Retrieves numerical values that correspond to the least and greatest queue priorities.

Returns in `*pMinPriority` and `*pMaxPriority` the numerical values that correspond to the least and greatest queue priorities respectively. Queue priorities follow a convention where lower numbers imply greater priorities. The range of meaningful queue priorities is given by [`*pMinPriority`, `*pMaxPriority`]. If you attempt to create a queue with a priority value that is out of the meaningful range as specfied by this API, the priority is automatically clamped down or up to either `*pMinPriority` or `*pMaxPriority` respectively. See *cnrtQueueCreateWithPriority* for details on creating a priority queue.

Parameters

- `[out] pMinPriority`: Pointer to an integer in which the numerical value for least queue priority is returned.
- `[out] pMaxPriority`: Pointer to an integer in which the numerical value for greatest queue priority is returned.

Return

- *cnrtSuccess*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtQueueCreateWithPriority* API.

4.1.20 cnrtDeviceResetPersistingL2Cache

cnrtRet_t `cnrtDeviceResetPersistingL2Cache`(void)

Resets all persisting cache lines in cache normal status.

Resets all persisting cache lines for current device. This API takes effect immediately once it is returned.

Return

- *cnrtSuccess*, *cnrtErrorNotSupport*

Note

- None.

Example

- None.

4.1.21 cnrtIpcGetNotifierHandle

cnrtRet_t `cnrtIpcGetNotifierHandle`(*cnrtIpcNotifierHandle* * *handle*, *cnrtNotifier_t* *notifier*)

Gets an interprocess handle for a previously allocated notifier. This handle may be copied into other processes and opened with *cnrtIpcOpenNotifierHandle*.

Parameters

- `[out] handle`: Pointer to a user allocated *cnrtIpcNotifierHandle* in which to return the event IPC handle.
- `[in] notifier`: The notifier handle created by calling *cnrtNotifierCreate* with *CNRT_NOTIFIER_INTERPROCESS* and *CNRT_NOTIFIER_DISABLE_TIMING_ALL*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorNoMem*, *cnrtErrorCndrvFuncCall*, *cnrtErrorNotSupport*

Note

- The `handle` will be invalid after you destroy the `notifier` by calling *cnrtNotifierDestroy*.
- The new notifier allocated by *cnrtIpcOpenNotifierHandle* may be invalid if `notifier` has been freed with *cnrtNotifierDestroy*.

4.1.22 cnrtIpcOpenNotifierHandle

cnrtRet_t `cnrtIpcOpenNotifierHandle`(*cnrtNotifier_t* **notifier*, *cnrtIpcNotifierHandle* *handle*)

Opens an interprocess notifier handle for use in the current process. This API returns a new notifier handle like a locally created event with *CNRT_NOTIFIER_DISABLE_TIMING_ALL* flag. This notifier must be freed with *cnrtNotifierDestroy*.

Parameters

- `[in] handle`: Interprocess handle got from *cnrtIpcGetNotifierHandle*.
- `[out] notifier`: A new notifier handle.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorNoMem*, *cnrtErrorCndrvFuncCall*, *cnrtErrorNotSupport*

Note

- The `notifier` which is allocated by *cnrtIpcOpenNotifierHandle* cannot use *cnrtIpcGetNotifierHandle* to get a new handle.

4.2 Execution Control Management

4.2.1 cnrtInvokeHostFunc

cnrtRet_t `cnrtInvokeHostFunc`(*cnrtQueue_t* *queue*, *cnrtHostFn_t* *fn*, void **userData*)

Enqueues a host function call in a queue.

This API is an asynchronous interface and *cnrtSuccess* does not represent the completion of host function execution. Call *cnrtQueueQuery()* or *cnrtQueueSync()* to confirm whether the host function call in the queue has been executed. The host function call must not perform any synchronization that may depend on other asynchronous tasks not mandated to run earlier, which can cause a deadlock. Without a mandated order, host function is executed (in independent queues) in undefined order and may be serialized.

Parameters

- `[in] queue`: Enqueued queue.
- `[in] fn`: The CPU API to call once preceding queue operations are complete.
- `[in] userData`: User-specified data to be passed to the API.

Return

- *cnrtSuccess*, *cnrtErrorInit*, *cnrtErrorCndrvFuncCall*

Note

- The host function call must not make any CNRT or CNDrv API call. Attempting to use CNRT and CNDrv API may result in *cnrtErrorCndrvFuncCall*.

Example

```
// test.mlu
void hostFunc1(void *args) {
  printf("%s: %p\n", __func__, args);
}
void hostFunc2(void *args) {
  printf("%s: %p\n", __func__, args);
}
__mlu_global__ kernel() {
  printf("%s\n", __func__);
}

int main() {
  void *arg1 = 0x1;
  void *arg2 = 0x2;

  // nullptr for default queue also works
  cnrtQueue_t queue;
  cnrtQueueCreate(&queue);

  // ...
  kernel<<<dim, cnrtFuncTypeBlock, queue>>>();

  cnrtInvokeHostFunc(queue, hostFunc1, arg1);
  cnrtInvokeHostFunc(queue, hostFunc2, arg2);
  cnrtQueueSync(queue);

  // print out:
  // kernel
  // hostFunc1: 1
  // hostFunc2: 2

  // ...
}
```

4.2.2 cnrtInvokeKernel

```
cnrtRet_t cnrtInvokeKernel(const void *kernel,
                           cnrtDim3_t dim,
                           cnrtFunctionType_t ktype,
                           void **args,
                           size_t reserved,
                           cnrtQueue_t queue)
```

Enqueues a device kernel in a queue.

Invokes a `kernel` function, and enqueues the kernel to the `queue`. If the kernel has N parameters, the `args` should point to array of N pointers. Each pointer, from args[0] to args[N - 1], pointer to the region of memory from which the actual parameter will be copied. This is an asynchronous API, the returned *cnrtSuccess* does not represent the result of the kernel execution.

Parameters

- `[in]` `kernel`: Device kernel symbol.
- `[in]` `dim`: The dimension of {x, y, z}.
- `[in]` `ktype`: The task size type.
- `[in]` `args`: The array of pointers to kernel parameter.
- `[in]` `reserved`: Reserved.
- `[in]` `queue`: The queue to invoke kernel.

Return

- *cnrtSuccess*, *cnrtErrorInit*, *cnrtErrorArgsInvalid*, *cnrtErrorNoDevice*, *cnrtErrorNoKernel*, *cnrtErrorNoModule*, *cnrtErrorNoCnrtContext*

Note

- None.

Example

```
// x.mlu
__mlu_entry void kernel(int a, int b) {
  printf("%d,%d\n", a, b);
}

int main() {
  cnrtQueue_t queue = NULL; // default queue
  cnrtDim3_t dim = {1, 1, 1};
  // The two ways to invoke kernel are equivalent.

  kernel<<<dim, cnrtFuncTypeBlock, queue>>>(1, 2);
  int a = 1, b = 2;
  void *args[] = {&a, &b};
  cnrtInvokeKernel(kernel, dim, cnrtFuncTypeBlock, args, 0, queue);

  cnrtQueueSync(queue);
}
```

4.3 Memory Management

4.3.1 cnrtMalloc

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

Allocates memory on the current device.

Allocates the `bytes` size of current MLU device memory, and returns a pointer `pPtr` to the allocated memory.

Parameters

- `[out] pPtr`: Pointer to allocated MLU memory.
- `[in] bytes`: Requested memory size in bytes.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorNoMem*, *cnrtErrorCndrvFuncCall*

Note

- Call *cnrtFree* to release the allocated memory, otherwise memory leak may occur.

Example

```
int main () {
  // Specify the memory size
  size_t size = sizeof(size_t) * N;
  // Allocate page-locked memory. For more information about the advantages of using␣
↪the cnrtHostMalloc,
  // see cnrtHostMalloc description.
  void *h = NULL;
  cnrtHostMalloc((void **)&h, size);

  // Initialize host memory allocated before
  ...

  // Allocate device memory
  void *d = NULL;
  cnrtMalloc((void **)&d, size);

  // Copy data from host to device
  cnrtMemcpy(d, h, size, cnrtMemcpyHostToDev);
  // Copy data from device to host
  cnrtMemcpy(h, d, size, cnrtMemcpyDevToHost);

  // Free resource
  cnrtFreeHost(h);
  cnrtFree(d);

  return 0;
}
```

4.3.2 cnrtMallocConstant

```
cnrtRet_t cnrtMallocConstant(void **pPtr,
                             size_t bytes)
```

Allocates constant memory on the current device.

Allocates the `bytes` size of current MLU device constant memory, and returns a pointer `pPtr` to the allocated memory.

Parameters

- `[out] pPtr`: Pointer to allocated MLU memory.
- `[in] bytes`: Requested memory size in bytes.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorNoMem*, *cnrtErrorCndrvFuncCall*

Note

- Call *cnrtFree* to release the allocated constant memory, otherwise memory leak may occur.

Example

```
int main () {
  // Specify the memory size
  size_t size = sizeof(size_t) * N;
  // Allocate page-locked memory. For more information about the advantages of using the cnrtHostMalloc,
  // see cnrtHostMalloc description.
  void *h = NULL;
  cnrtHostMalloc((void **)&h, size);

  // Initialize host memory allocated before
  ...

  // Allocate device memory
  void *d = NULL;
  cnrtMallocConstant((void **)&d, size);

  // Copy data from host to device
  cnrtMemcpy(d, h, size, cnrtMemcpyHostToDev);
  // Copy data from device to host
  cnrtMemcpy(h, d, size, cnrtMemcpyDevToHost);

  // Free resource
  cnrtFreeHost(h);
  cnrtFree(d);

  return 0;
}
```

4.3.3 cnrtFree

cnrtRet_t `cnrtFree`(void **ptr*)

Frees the memory on the device.

Frees the MLU device memory pointed by `ptr`, which must have been returned by a previous call to *cnrtMalloc*. Otherwise, if *cnrtFree*(`ptr`) has already been called before, an error is returned. If `ptr` is 0, no operation is performed.

Parameters

- `[in] ptr`: Pointer to the device memory to be freed.

Return

- *cnrtSuccess*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtMalloc*.

4.3.4 cnrtHostMalloc

cnrtRet_t `cnrtHostMalloc`(void ***pPtr*,
size_t *bytes*)

Allocates page-locked memory on the host.

Allocates size of `bytes` of page-locked host memory, and returns a pointer `pPtr` to the allocated memory. The memory ranges allocated with this API can be tracked by the driver so that the API calls such as *cnrtMemcpy* are accelerated automatically. In comparison with pageable memory requested by the system malloc function, the page-locked memory has better read and write performance. However, allocating large amount of page-locked memory with this API may lead to lower performance due to reduction of available amount of memory for paging. So for best practice, it is recommended to use this API to allocate staging areas for data exchange between host and the MLU device.

Parameters

- `[out] pPtr`: Pointer to the allocated host memory.
- `[in] bytes`: The size requested to allocate.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorSysNoMem*, *cnrtErrorCndrvFuncCall*

Note

- Call *cnrtFreeHost* to release the allocated host memory, otherwise the memory leak may occur.

Example

- See example in *cnrtMalloc* API.

4.3.5 cnrtFreeHost

cnrtRet_t `cnrtFreeHost`(void **ptr*)

Frees the page-locked memory.

Frees the host memory pointed by `ptr`, this API is only used to free the host memory that is allocated by the *cnrtHostMalloc*.

Parameters

- `[in] ptr`: Pointer to the host memory.

Return

- *cnrtSuccess*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtMalloc* API.

4.3.6 cnrtMemcpy

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

Copies data from source address to destination address.

Synchronously copies the size of `bytes` bytes of data from the source address pointed by `src` to the destination address pointed by `dst` with the copy direction `dir`.

Parameters

- `[in] dst`: Pointer to the destination address.
- `[in] src`: Pointer to the source address.
- `[in] bytes`: The memory size in bytes to be copied.
- `[in] dir`: Data copying direction.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorCndrvFuncCall*

Note

- This API is used to copy data synchronously. To copy data asynchronously, call *cnrtMemcpyAsync*.

Example

- See example in *cnrtMalloc* API.

4.3.7 cnrtMemcpy2D

```
cnrtRet_t cnrtMemcpy2D(void *dst,
                       size_t dpitch,
                       const void *src,
                       size_t spitch,
                       size_t width,
                       size_t height,
                       cnrtMemTransDir_t direction)
```

Uses 2D to copy data from source address to destination address.

Synchronously reads `spitch * width` bytes of data from the source address pointed by `src`, and writes it to the `dpitch * width` bytes of destination address pointed by `dst` with the copy direction `dir`. The `dir` must be cnrtMemcpyDevToDev or cnrtMemcpyNoDirection.

Parameters

- `[in] dst`: Pointer to the destination address.
- `[in] dpitch`: The pitch of the destination memory. It cannot be less than the width, or greater than 4MB.
- `[in] src`: Pointer to the source address.
- `[in] spitch`: The pitch of the source memory. It cannot be less than the width, or greater than 4MB.
- `[in] width`: The width of the memory to be copied. It cannot be greater than 1MB.
- `[in] height`: The height of the memory to be copied. It cannot be greater than 1MB.
- `[in] direction`: Data copy direction. It must be cnrtMemcpyDevToDev or cnrtMemcpyNoDirection.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorCndrvFuncCall*

Note

- It only supports CE platform currently.
- None of the parameters can be 0. The size (width * height) of the data transfer cannot be greater than 16MB.

Example

```
// ...
// Specify the memory size
size_t size = dpitch * spitch;

// Allocate device memory of the source memory
void *src = NULL;
cnrtMalloc((void **)&src, size);
```

```
// Initialize src memory allocated before
...

// Allocate device memory of the destination memory
void *dst = NULL;
cnrtMalloc((void **)&dst, size);

// Initialize param of dpitch, spitch, width, height
...

// Use 2D to copy data from src to dst
cnrtMemcpy2D(dst, dpitch, src, spitch, width, height, cnrtMemcpyDevToDev);

// Free resource
cnrtFree(src);
cnrtFree(dst);
// ...
```

4.3.8 cnrtMemcpy3D

cnrtRet_t `cnrtMemcpy3D(const` *cnrtMemcpy3dParam_t* **p*)

Uses 3D to copy data from source address to destination address.

Synchronously reads `src_size` bytes of data from the source address pointed by `p->src` or `p->srcPtr.ptr`, and writes it to the `dst_size` bytes of data from the destination address pointed by `p->dst` or `p->dstPtr.ptr`. The `src_size` is configured by `p->extent` and `p->srcPtr`, the `dst_size` is configured by `p->extent` and `p->dstPtr`. The copy direction `direction` must be in *cnrtMemTransDir_t*. The `direction` must be *cnrtMemcpyDevToDev* or *cnrtMemcpyNoDirection*.

Parameters

- `[in] p`: Pointer to the 3D memory copy parameter.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorCndrvFuncCall*

Note

- It only supports CE platform currently.
- The size (p->extent.width * p->extent.height * pst->extent.depth) of the data transfer cannot be greater than 16MB.

Example

```
// ...
cnrtMemcpy3dParam_t *p;
p = (cnrtMemcpy3dParam_t *)malloc(sizeof(*p));

// Initialize 3D param of p
...

// Specify the memory size
size_t size = dstPtr.pitch * srcPtr.pitch * extent.depth;

// Allocate device memory of the source memory
void *src = NULL;
cnrtMalloc((void **)&src, size);

// Initialize src memory allocated before
...

// Allocate device memory of the destination memory
void *dst = NULL;
cnrtMalloc((void **)&dst, size);

p->dstPtr.ptr = dst;
p->srcPtr.ptr = src;

// Use 3D to copy data from src to dst
cnrtMemcpy3D(p);

// Free resource
cnrtFree(src);
cnrtFree(dst);
free(p);
// ...
```

4.3.9 cnrtMemcpyAsync

```
cnrtRet_t cnrtMemcpyAsync(void *dst,
                          void *src,
                          size_t bytes,
                          cnrtQueue_t queue,
                          cnrtMemTransDir_t dir)
```

Copies data from source address to destination address asynchronously.

Asynchronously copies the size of `bytes` bytes of data from the source address pointed by `src` to the destination address pointed by `dst` with the copy direction `dir`. The `dir` must be one of *cnrtMemcpyHostToDev*, *cnrtMemcpyDevToDev*, *cnrtMemcpyDevToHost*, *cnrtMemcpyPeerToPeer*, and *cnrtMemcpyNoDirection*, in *cnrtMemTransDir_t*.

Parameters

- `[in]` `dst`: Pointer to the destination address.
- `[in]` `src`: Pointer to the source address.
- `[in]` `bytes`: The memory size to be copied in bytes.
- `[in]` `dir`: The data copying direction.
- `[in]` `queue`: The queue handle created by calling *cnrtQueueCreate*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorCndrvFuncCall*

Note

- This API is used to copy data asynchronously. To copy data synchronously, call *cnrtMemcpy*.

Example

```
int main () {
  // Prepare input and output
  size_t size = sizeof(size_t) * N;
  char *h0 = NULL;
  char *h1 = NULL;
  cnrtHostMalloc((void **)&h0, size);
  cnrtHostMalloc((void **)&h1, size);
  memset(h0, 'a', size);

  void *d = NULL;
  cnrtMalloc((void **)&d, size);

  // Create queue
```

```
    cnrtQueue_t queue;
    cnrtQueueCreate(&queue);

    // Memcpy Async
    cnrtMemcpyAsync(d, h0, size, queue, cnrtMemcpyHostToDev);
    cnrtMemcpyAsync(h1, d, size, queue. cnrtMemcpyDevToHost);
    cnrtQueueSync(queue);

    // Free resource
    cnrtQueueDestroy(queue);
    cnrtFreeHost(h0);
    cnrtFreeHost(h1);
    cnrtFree(d);

    return 0;
  }
```

4.3.10 cnrtMemcpyAsync_V2

cnrtRet_t `cnrtMemcpyAsync_V2`(void **dst*,
void **src*,
size_t *bytes*,
cnrtQueue_t *queue*,
cnrtMemTransDir_t *dir*)

Copies data from source address to destination address asynchronously.

Asynchronously copies the size of `bytes` bytes of data from the source address pointed by `src` to the destination address pointed by `dst` with the copy direction `dir`. The `dir` must be one of *cnrtMemcpyHostToDev*, *cnrtMemcpyDevToDev*, *cnrtMemcpyDevToHost*, *cnrtMemcpyPeerToPeer*, and *cnrtMemcpyNoDirection*, in *cnrtMemTransDir_t*.

Parameters

- `[in]` `dst`: Pointer to the destination address.
- `[in]` `src`: Pointer to the source address.
- `[in]` `bytes`: The memory size to be copied in bytes.
- `[in]` `dir`: The data copying direction.
- `[in]` `queue`: The queue handle created by calling *cnrtQueueCreate*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorCndrvFuncCall*

Note

- This API exhibits asynchronous behavior for most use cases.

Example

- See example in *cnrtMemcpyAsync*.

4.3.11 cnrtMemcpyPeer

```
cnrtRet_t cnrtMemcpyPeer(void *dst,
                         int dstDev,
                         void *src,
                         int srcDev,
                         size_t bytes)
```

Copies data between two devices.

Synchronously copies `bytes` of data from the address pointed by `src` in source device ordinal `srcDev` to the address pointed by `dst` in destination device ordinal `dstDev`.

Parameters

- `[in] dst`: Pointer to the destination address.
- `[in] dstDev`: The destination device ordinal.
- `[in] src`: Pointer to the source address.
- `[in] srcDev`: The source device ordinal.
- `[in] bytes`: The memory size to be copied in bytes.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorDeviceInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorCndrvFuncCall*

Note

- Devices support peer-to-peer communications. If the two MLU devices are not peerable, it may be hardware problems.
- The two devices must be peerable when calling *cnrtMemcpyPeer* to copy data. Call *cnrtGetPeerAccessibility* to check if the two devices are peerable.

Example

```
int main () {
  unsigned int count = 0;
  cnrtGetDeviceCount(&count);
  if (count < 2) {
    printf("Warning: Two or more MLU devices are required for Peer-to-Peer.\n");
    return 0;
  }
  // To simplify sample code, it is supposed that srcDev is 0, and dstDev is 1
  int srcDev = 0;
  int dstDev = 1;

  int canAccess = -1;
  cnrtGetPeerAccessibility(&canAccess, srcDev, dstDev);
  if (canAccess != 1) {
      printf("Error: There is no p2p accessibility MLU devices in the current system.\n");
      return -1;
  }

  cnrtSetDevice(srcDev);
  char *src = NULL;
  cnrtMalloc((void **)&src, size);
  cnrtMemset(src, 'a', size);

  cnrtSetDevice(dstDev);
  char *dst = NULL;
  cnrtMalloc((void **)&dst, size);

  cnrtMemcpyPeer(dst, dstDev, src, srcDev, size);

  // Free resource
  cnrtFree((void *)src);
  cnrtFree((void *)dst);

  return 0;
}
```

4.3.12 cnrtMemcpyPeerAsync

cnrtRet_t **cnrtMemcpyPeerAsync**(void **dst*,
int *dstDev*,
void **src*,
int *srcDev*,
size_t *bytes*,
cnrtQueue_t *queue*)

Copies data between two devices asynchronously.

Asynchronously copies `bytes` of data from the address pointed by `src` in source device ordinal `srcDev` to the address pointed by `dst` in destination device ordinal `dstDev`.

Parameters

- `[in] dst`: Pointer to the destination address.
- `[in] dstDev`: The destination device ordinal.
- `[in] src`: Pointer to the source address.
- `[in] srcDev`: The source device ordinal.
- `[in] queue`: The queue handle created by calling *cnrtQueueCreate*.
- `[in] bytes`: The memory size to be copied in bytes.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorDeviceInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorCndrvFuncCall*

Note

- This API is used to copy data asynchronously. To copy data synchronously, call *cnrtMemcpyPeer*.
- Devices support peer-to-peer communications. If the two MLU devices are not peerable, it may be hardware problems.
- The two devices must be peerable when calling *cnrtMemcpyPeerAsync* to copy data. Call *cnrtGetPeerAccessibility* to check if the two devices are peerable.

Example

```
int main () {
  unsigned int count = 0;
  cnrtGetDeviceCount(&count);
  if (count < 2) {
    printf("Warning: Two or more MLU devices are required for Peer-to-Peer.\n");
    return 0;
  }

  // To simplify sample code, suppose sdev is 0, ddev is 1
  int srcDev = 0;
  int dstDev = 1;

  int canAccess = -1;
  cnrtGetPeerAccessibility(&canAccess, srcDev, dstDev);
  if (canAccess != 1) {
      printf("Error: There is no P2P accessibility MLU devices in the current system.
↪\n");
      return -1;
  }
  cnrtSetDevice(srcDev);
  char *src = NULL;
  cnrtMalloc((void **)&src, size);

  cnrtQueue_t queue;
  cnrtQueueCreate(&queue);
  cnrtMemsetAsync(src, 'a', size, queue);

  cnrtSetDevice(dstDev);
  char *dst = NULL;
  cnrtMalloc((void **)&dst, size);

  cnrtMemcpyPeer(dst, dstDev, src, srcDev, size, queue);
  cnrtQueueSync(queue);

  // Free resource
  cnrtFree((void *)src);
  cnrtFree((void *)dst);
  cnrtQueueDestroy(queue);

  return 0;
}
```

4.3.13 cnrtGetSymbolAddress

cnrtRet_t **cnrtGetSymbolAddress**(void ***pPtr*,
const void **symbol*)

Finds the address of an MLU symbol.

Returns in `pPtr` the address of `symbol` on device.

Parameters

- `[out] pPtr`: Pointer to the address of the `symbol` on device.
- `[in] symbol`: Symbol to query.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*, *cnrtErrorInvalidSymbol*

Example

```
// __mlu_entry__ function and __mlu_device__ variables must be written in a single␣
↪file named *.mlu
#define TEST_SIZE 64
__mlu_device__ int array_f[TEST_SIZE];
__mlu_entry__ void kernelSetDeviceMem(size_t size) {
  int i = 0;
  for (i = 0; i < size; i++) {
    array_f[i] = 0x12345678;
  }
}

// Hybrid programming
int main () {
  size_t size = 0;
  void *addr = NULL;
  cnrtDim3_t dim = {1, 1, 1};
  cnrtQueue_t queue;
  CNRT_CHECK(cnrtQueueCreate(&queue));
  CNRT_CHECK(cnrtGetSymbolAddress(&addr, array_f));
  CNRT_CHECK(cnrtGetSymbolSize(&size, array_f));
  int _h_array[TEST_SIZE];
  int i = 0;
  for (i = 0; i < TEST_SIZE;i++) {
    _h_array[i] = 0;
  }
  hostSetDeviceMem(TEST_SIZE, dim, cnrtFuncTypeBlock, queue);
  CNRT_CHECK(cnrtGetLastError());
  CNRT_CHECK(cnrtQueueSync(queue));
  CNRT_CHECK(cnrtMemcpyFromSymbol(_h_array, (const void *)&array_f, TEST_SIZE *␣
↪sizeof(int), 0, cnrtMemcpyNoDirection));
  for (i = 0; i < TEST_SIZE; i++) {
    printf("After copy from symbol, array[%d] is %x.\n", i, _h_array[i]);
  }
  cnrtQueueDestroy(queue);
}
```

4.3.14 cnrtGetSymbolSize

```
cnrtRet_t cnrtGetSymbolSize(size_t *size,
                            const void *symbol)
```

Finds the size of an MLU symbol.

Returns in `size` the size of `symbol` on the device.

Parameters

- `[out] size`: Pointer to the size of the `symbol` on the device.
- `[in] symbol`: Symbol to query.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*, *cnrtErrorInvalidSymbol*

Example

- See example in *cnrtGetSymbolAddress* API.

4.3.15 cnrtMemcpyFromSymbol

```
cnrtRet_t cnrtMemcpyFromSymbol(void *dst,
                               const void *symbol,
                               size_t bytes,
                               size_t offset,
                               cnrtMemTransDir_t dir)
```

Copies data from the given symbol on the device.

Copies `bytes` data from the memory area pointed to by `offset` from the start of `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap.

Parameters

- `[in] dst`: Pointer to the destination address.
- `[in] symbol`: Symbol address on the host.
- `[in] bytes`: The memory size to be copied in bytes.
- `[in] offset`: Offset from start of symbol in bytes.
- `[in] dir`: Data copy direction. It must be *cnrtMemcpyDevToHost*, *cnrtMemcpyPeerToPeer* , *cnrtMemcpyDevToDev* or *cnrtMemcpyNoDirection*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorInvalidSymbol*

Note

- This API is used to copy data synchronously. To copy data asynchronously, call *cnrtMemcpyFromSymbolAsync*.

Example

- See example in *cnrtGetSymbolAddress* API.

4.3.16 cnrtMemcpyToSymbol

```
cnrtRet_t cnrtMemcpyToSymbol(const void *symbol,
                             const void *src,
                             size_t bytes,
                             size_t offset,
                             cnrtMemTransDir_t dir)
```

Copies data to the given symbol on the device.

Copies `bytes` data from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of `symbol`. The memory areas may not overlap.

Parameters

- `[in] symbol`: Symbol address on the host.
- `[in] src`: Pointer to the source address.
- `[in] bytes`: The memory size to be copied in bytes.
- `[in] offset`: Offset from start of symbol in bytes.
- `[in] dir`: Data copy direction. It must be *cnrtMemcpyHostToDev*, *cnrtMemcpyPeerToPeer* , *cnrtMemcpyDevToDev* or *cnrtMemcpyNoDirection*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorInvalidSymbol*

Note

- This API is used to copy data synchronously. To copy data asynchronously, call *cnrtMemcpyToSymbolAsync*.

Example

- None.

4.3.17 cnrtMemcpyFromSymbolAsync

```
cnrtRet_t cnrtMemcpyFromSymbolAsync(void *dst,
                                    const void *symbol,
                                    size_t bytes,
                                    size_t offset,
                                    cnrtMemTransDir_t dir,
                                    cnrtQueue_t queue)
```

Copies data asynchronously from the given symbol on the device.

Copies `bytes` data asynchronously from the memory area pointed to by `offset` from the start of `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap.

Parameters

- `[in] dst`: Pointer to the destination address.
- `[in] symbol`: Symbol address on the host.
- `[in] bytes`: The memory size to be copied in bytes.
- `[in] offset`: Offset from start of symbol in bytes.
- `[in] dir`: Data copy direction. It must be *cnrtMemcpyDevToHost*, *cnrtMemcpyPeerToPeer* , *cnrtMemcpyDevToDev* or *cnrtMemcpyNoDirection*.
- `[in] queue`: The queue handle created by calling *cnrtQueueCreate*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorInvalidSymbol*

Note

- This API is used to copy data asynchronously. To copy data synchronously, call *cnrtMemcpyFromSymbol*.

Example

- None.

4.3.18 cnrtMemcpyFromSymbolAsync_V2

```
cnrtRet_t cnrtMemcpyFromSymbolAsync_V2(void *dst,
                                       const void *symbol,
                                       size_t bytes,
                                       size_t offset,
                                       cnrtMemTransDir_t dir,
                                       cnrtQueue_t queue)
```

Copies data asynchronously from the given symbol on the device.

Copies `bytes` data asynchronously from the memory area pointed to by `offset` from the start of `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap.

Parameters

- `[in] dst`: Pointer to the destination address.
- `[in] symbol`: Symbol address on the host.
- `[in] bytes`: The memory size to be copied in bytes.
- `[in] offset`: Offset from start of symbol in bytes.
- `[in] dir`: Data copy direction. It must be *cnrtMemcpyDevToHost*, *cnrtMemcpyPeerToPeer* , *cnrtMemcpyDevToDev* or *cnrtMemcpyNoDirection*.
- `[in] queue`: The queue handle created by calling *cnrtQueueCreate*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorInvalidSymbol*

Note

- This API exhibits asynchronous behavior for most use cases.

Example

- None.

4.3.19 cnrtMemcpyToSymbolAsync

cnrtRet_t `cnrtMemcpyToSymbolAsync(const` void **symbol*, `const` void **src*, size_t *bytes*, size_t *offset*, *cnrtMemTransDir_t* *dir*, *cnrtQueue_t* *queue*)

Copies data asynchronously to the given symbol on the device.

Copies `bytes` data asynchronously from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of `symbol`. The memory areas may not overlap.

Parameters

- `[in] symbol`: Symbol address on the host.
- `[in] src`: Pointer to the source address.
- `[in] bytes`: The memory size to be copied in bytes.
- `[in] offset`: Offset from start of symbol in bytes.
- `[in] dir`: Data copy direction. It must be *cnrtMemcpyHostToDev*, *cnrtMemcpyPeerToPeer* , *cnrtMemcpyDevToDev* or *cnrtMemcpyNoDirection*.
- `[in] queue`: The queue handle created by calling *cnrtQueueCreate*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorInvalidSymbol*

Note

- This API is used to copy data asynchronously. To copy data synchronously, call *cnrtMemcpyToSymbol*.

Example

- None.

4.3.20 cnrtMemcpyToSymbolAsync_V2

```
cnrtRet_t cnrtMemcpyToSymbolAsync_V2(const void *symbol,
                                     const void *src,
                                     size_t bytes,
                                     size_t offset,
                                     cnrtMemTransDir_t dir,
                                     cnrtQueue_t queue)
```

Copies data asynchronously to the given symbol on the device.

Copies `bytes` data asynchronously from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of `symbol`. The memory areas may not overlap.

Parameters

- `[in] symbol`: Symbol address on the host.
- `[in] src`: Pointer to the source address.
- `[in] bytes`: The memory size to be copied in bytes.
- `[in] offset`: Offset from start of symbol in bytes.
- `[in] dir`: Data copy direction. It must be *cnrtMemcpyHostToDev*, *cnrtMemcpyPeerToPeer* , *cnrtMemcpyDevToDev* or *cnrtMemcpyNoDirection*.
- `[in] queue`: The queue handle created by calling *cnrtQueueCreate*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorInvalidSymbol*

Note

- This API exhibits asynchronous behavior for most use cases.

Example

- None.

4.3.21 cnrtMemGetInfo

cnrtRet_t `cnrtMemGetInfo`(size_t **free*,
size_t **total*)

Gets the free and total device memory.

Returns in `total` the total amount of memory available to the current device. Returns in `free` the amount of memory on the device that is free according to the OS.

Parameters

- `[out] free`: Pointer to the free memory in bytes.
- `[out] total`: Pointer to the total memory in bytes.

Return

- *cnrtSuccess*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- It is not guaranteed that all of the memory that OS reports as free can be allocated by user.

Example

```
int main () {
  size_t available, total;
  cnrtMemGetInfo(&available, &total);
  printf("free: %#lx, total: %#lx.\n", available, total);
  return 0;
}
```

4.3.22 cnrtMemset

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

Initializes or sets device memory to a value.

Synchronously fills the first `bytes` of the memory area pointed to by `ptr` with constant byte value `value`.

Parameters

- `[in] ptr`: Pointer to the device address to be set.
- `[in] value`: The value to set.
- `[in] bytes`: Size in bytes to set.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- This API is used to set data synchronously. To set data asynchronously, call *cnrtMemsetAsync*.

Example

- See example in *cnrtMalloc* API.

4.3.23 cnrtMemsetAsync

```
cnrtRet_t cnrtMemsetAsync(void *ptr,
                          int value,
                          size_t bytes,
                          cnrtQueue_t queue)
```

Initializes or sets device memory to a value.

Asynchronously fills the first `bytes` of the memory area pointed to by `ptr` with constant byte value `value`.

Parameters

- `[in] ptr`: Pointer to the device address to be set.
- `[in] value`: The value to set.
- `[in] bytes`: Size in bytes to set.
- `[in] queue`: The queue.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- This API is used to set data asynchronously. To set data synchronously, call *cnrtMemset*.
- The `value` is set for each byte.

Example

- See example in *cnrtMemcpyPeerAsync* API.

4.3.24 cnrtPointerGetAttributes

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

Gets the attributes of a specified pointer.

Gets attributes of `ptr` in `attr`. See "Cambricon BANG C/C++ Programming Guide" for details.

Parameters

- [out] attr: Pointer to *cnrtPointerAttributes_t*.
- [in] ptr: A specified pointer.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorNotSupport*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

```
int main () {
  cnrtPointerAttributes_t attributes;
  size_t size = sizeof(size_t) * N;
  void *dev_addr = NULL, *cpu_addr = NULL;

  cnrtMalloc(&dev_addr, size);
  cnrtMmap(dev_addr, &cpu_addr, size);

  // UVA without offset
  cnrtPointerGetAttributes(&attributes, cpu_addr);
  if (attributes.devicePointer != dev_addr) {
    printf("cnrtPointerGetAttributes failed.\n");
    return -1;
  }

  // UVA with offset
  cnrtPointerGetAttributes(&attributes, (void *)((unsigned long)cpu_addr + 0x100));
  if (attributes.devicePointer != (void *)((unsigned long)dev_addr + 0x100)) {
    printf("cnrtPointerGetAttributes failed.\n");
    return -1;
  }

  cnrtMunmap(cpu_addr, size);
  cnrtFree(dev_addr);

  return 0;
}
```

4.3.25 cnrtMcacheOperation

```
cnrtRet_t cnrtMcacheOperation(void *ptr,
                              void *hostPtr,
                              size_t size,
                              cnrtCacheOps_t ops)
```

Flushes or invalidates cache on the host.

Ensures cache consistency if the `hostPtr` is cached. Specifies the operation on cache with *cnrtCacheOps_t* enum `ops`. Currently, only the flush operation is supported.

Parameters

- `[in]` `ptr`: Reserved for checking the device address legality.
- `[in]` `hostPtr`: Host memory address to do cache operation..
- `[in]` `size`: The number of bytes to do cache operation.
- `[in]` `ops`: The operation type on cache.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- This API first makes `hostPtr` be aligned to cache line. The `size` may change to be aligned to cache line. The cache line now is 64-byte in driver.

Example

- See example in *cnrtMmapCached*.

4.3.26 cnrtMmap

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

Maps the range of device memory into the user-mode uncached virtual address.

Maps memory address of the device address pointer `ptr` into the user-mode address, and returns the uncached host memory address pointer `pHostPtr`.

Parameters

- `[in]` `ptr`: The device address to map.
- `[out]` `pHostPtr`: Pointer to the host memory address to be mapped into.
- `[in]` `size`: The size of the memory in bytes to be mapped into.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- The output parameter `pHostPtr` is a pointer to uncached host address. it is not necessary to ensure cache consistency, but this may impact performance.

Example

```
int main () {
  void *addr = NULL;
  void *host = NULL;
  size_t size = 0x100000;

  cnrtMalloc(&addr, size);
  cnrtMmap(addr, &host, size);

  memset(host, 'a', size);

  void *host1 = NULL;
  cnrtMmap(addr, &host1, size);
  memset(host1, 'b', size);

  void *cmp = malloc(size);
  cnrtMemcpy(cmp, addr, size, cnrtMemcpyNoDirection);

  if (memcmp(cmp, host1, size)) printf("memcmp failed!\n");

  free(cmp);
  cnrtMunmap(host1, size);
  cnrtFree(addr);

  return 0;
}
```

4.3.27 cnrtMmapCached

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

Maps the range of device memory address into the cached host address space.

Maps memory address of the device address pointer `ptr` into the host memory space, and returns the cached host address pointer `pHostPtr`.

Parameters

- `[in] ptr`: The device address to map.
- `[out] pHostPtr`: Pointer to the host address to be mapped into.
- `[in] size`: The size of the memory in bytes to be mapped into.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- The output parameter `pHostPtr` is a pointer to cached host address, so a cache flush operation is required to ensure cache consistency.

Example

```
int main () {
  void *addr = NULL;
  void *host = NULL;
  size_t size = 0x100000;

  cnrtMalloc(&addr, size);
  cnrtMmapCached(addr, &host, size);
  memset(host, 'a', size);
  cnrtMcacheOperation(addr, host, size, CNRT_FLUSH_CACHE);

  void *host1 = NULL;
  cnrtMmapCached(addr, &host1, size);
  memset(host1, 'b', size);
  cnrtMcacheOperation(addr, host, size, CNRT_FLUSH_CACHE);

  void *cmp = malloc(size);
  cnrtMemcpy(cmp, addr, size, cnrtMemcpyNoDirection);

  if (memcmp(cmp, host1, size)) printf("memcmp failed!\n");

  free(cmp);
  cnrtMunmap(host1, size);
  cnrtFree(addr);

  return 0;
}
```

4.3.28 cnrtMunmap

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

Unmaps the host address.

Unmaps the mapped host address and device memory address. The mapping is created by *cnrtMmap* or *cnrtMmapCached*.

Parameters

- `[in] hostPtr`: Host address to be unmapped.
- `[in] size`: The size of the host address to be unmapped.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtMmapCached*.

4.4 Notifier Management

4.4.1 cnrtNotifierCreate

```
cnrtRet_t cnrtNotifierCreate(cnrtNotifier_t *pNotifier)
```

Creates a notifier for the current device.

Returns a pointer `notifier` to the newly created notifier. For more information about notifier, see "Cambricon BANG C/C++ Programming Guide".

Parameters

- `[out] pNotifier`: Pointer to the newly created notifier.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- Call *cnrtNotifierDestroy* to release the notifier memory; otherwise the memory leaks may occur.
- To learn more about queue life cycle management, see "Cambricon CNDrv Developer Guide".

Example

- See example in *cnrtNotifierElapsedTime*.

4.4.2 cnrtNotifierCreateWithFlags

cnrtRet_t `cnrtNotifierCreateWithFlags`(*cnrtNotifier_t* **pNotifier*, unsigned int *flags*)

Creates a notifier with flags for the current device.

Returns a pointer `notifier` to the newly created notifier. The flags that can be specified include *CNRT_NOTIFIER_DEFAULT*, *CNRT_NOTIFIER_DISABLE_TIMING_SW*, *CNRT_NOTIFIER_DISABLE_TIMING_ALL*, *CNRT_NOTIFIER_INTERPROCESS*. For more information about notifier, see "Cambricon BANG C/C++ Programming Guide".

Parameters

- `[out] pNotifier`: Pointer to the newly created notifier.
- `[in] flags`: notifier creation flags.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- Call *cnrtNotifierDestroy* to release the notifier memory; otherwise the memory leaks may occur.
- To learn more about queue life cycle management, see "Cambricon CNDrv Developer Guide".

Example

- See example in *cnrtNotifierElapsedTime*.

4.4.3 cnrtNotifierDestroy

cnrtRet_t `cnrtNotifierDestroy`(*cnrtNotifier_t* *notifier*)

Destroys a notifier that is created by *cnrtNotifierCreate*.

Destroys a notifier pointed by `notifier`. For more information about notifier, see "Cambricon BANG C/C++ Programming Guide".

Parameters

- `[in] notifier`: The notifier that is created by *cnrtNotifierCreate*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- If a notifier is placed into a queue, but it is not completed when the *cnrtNotifierDestroy* API is called, the *cnrtNotifierDestroy* API will return immediately; but the resources associated with this notifier will be released automatically only after the notifier is completed.

Example

- See example in *cnrtNotifierElapsedTime*.

4.4.4 cnrtNotifierElapsedTime

```
cnrtRet_t cnrtNotifierElapsedTime(cnrtNotifier_t start,
                                  cnrtNotifier_t end,
                                  float *ms)
```

Computes the software time duration between the starting and ending of notifiers.

Computes the software time duration between the starting notifier `start` and the ending notifier `end`. This API is used to measure the execution time of all the tasks between the starting notifier and ending notifier. The measurement that can be used to improve the performance.

Parameters

- `[in] start`: The handle of the starting notifier created by the *cnrtNotifierCreate* API.
- `[in] end`: The handle of ending notifier created by the *cnrtNotifierCreate* API.
- `[out] ms`: The software time duration between the starting and ending of notifiers in ms.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorBusy*, *cnrtErrorCndrvFuncCall*

Note

- Call *cnrtPlaceNotifier* to place the starting and ending notifiers into the queue respectively first. Otherwise, *cnrtErrorCndrvFuncCall* is returned.
- If the *cnrtNotifierElapsedTime* API is called before the starting notifier or ending notifier is placed into the queue, *cnrtErrorCndrvFuncCall* is returned.

Example

```
int main () {
  int ret;

  // Create notifier
  cnrtNotifier_t notifier_s;
  cnrtNotifier_t notifier_e;
  cnrtNotifier_t notifier_dis_tim;
```

```
cnrtNotifierCreate(&notifier_s);
cnrtNotifierCreate(&notifier_e);
//we can not call cnrtNotifierElapsedTime if specify CNRT_NOTIFIER_DISABLE_TIMING_ALL.
cnrtNotifierCreateWithFlags(&notifier_dis_tim, CNRT_NOTIFIER_DISABLE_TIMING_ALL);

// Create queue
cnrtQueue_t queue;
cnrtQueueCreate(&queue);

size_t size = 0x1000000;
char *host_mem = (char *)malloc(size);
void *dev_mem = NULL;
cnrtMalloc(&dev_mem, size);

// Place notifier into a queue
cnrtPlaceNotifier(notifier_s, queue);
// Push a task into queue between the two notifiers
cnrtMemcpyAsync(dev_mem, host_mem, size, queue, cnrtMemcpyHostToDev);
cnrtPlaceNotifier(notifier_e, queue);
cnrtPlaceNotifier(notifier_dis_tim, queue);

cnrtQueueSync(queue);

// Query notifier
cnrtQueryNotifier(notifier_s);
cnrtQueryNotifier(notifier_e);
cnrtQueryNotifier(notifier_dis_tim);

// Wait for notifier
cnrtWaitNotifier(notifier_s);
cnrtWaitNotifier(notifier_e);
cnrtWaitNotifier(notifier_dis_tim);

// Compute the software duration
float ms;
cnrtNotifierElapsedTime(notifier_s, notifier_e, &ms);
printf("software time consuming between the two notifier is %f\n", ms);

cnrtNotifierDestroy(notifier_s);
cnrtNotifierDestroy(notifier_e);
cnrtNotifierDestroy(notifier_dis_tim);
```

```
  cnrtQueueDestroy(queue);
  cnrtFree(dev_mem);
  free(host_mem);

  return 0;
}
```

4.4.5 cnrtQueryNotifier

```
cnrtRet_t cnrtQueryNotifier(cnrtNotifier_t notifier)
```

Queries the status of notifier in a queue.

Returns *cnrtSuccess* if the `notifier` in the queue is completed. Returns *cnrtErrorBusy* if the `notifier` is still executing.

Parameters

- `[in] notifier`: The handle of the notifier to query.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorBusy*, *cnrtErrorCndrvFuncCall*

Note

- When querying the status of the notifier, if you call *cnrtPlaceNotifier* again on the same notifier, the query will be based on the most recent call to *cnrtPlaceNotifier*. The result of the previous query will be overwritten.

Example

- See example in *cnrtNotifierElapsedTime* API.

4.4.6 cnrtPlaceNotifier

```
cnrtRet_t cnrtPlaceNotifier(cnrtNotifier_t notifier,
                            cnrtQueue_t queue)
```

Places a notifier into a specified `queue`.

The notifier can be used to measure the execution time of all the tasks.

Parameters

- `[in] notifier`: The handle of the notifier to be placed into the queue. Create the notifier by calling the *cnrtNotifierCreate* API.

- `[in] queue`: The queue in which the notifier is placed. Create the queue by calling *cnrtQueueCreate*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtNotifierElapsedTime* API.

4.4.7 cnrtWaitNotifier

cnrtRet_t `cnrtWaitNotifier`(*cnrtNotifier_t notifier*)

Waits for a notifier to be completed.

Waits for a `notifier` in the queue to be completed before executing all future tasks in this queue. The `notifier` is the most recent one called by the *cnrtPlaceNotifier* API in the queue. Returns *cnrtSuccess* if the notifier is completed.

Parameters

- `[in] notifier`: The handle of the notifier to be waited for.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorCndrvFuncCall*

Note

- If *cnrtPlaceNotifier* has not been called on the notifier before calling this API, *cnrtSuccess* will be returned.
- This API is used for synchronization in a single queue. To synchronize across queues, use *cnrtQueueWaitNotifier*.

Example

- See example in *cnrtNotifierElapsedTime*.

4.4.8 cnrtNotifierDuration

cnrtRet_t `cnrtNotifierDuration`(*cnrtNotifier_t start*,
cnrtNotifier_t end,
float **us*)

Computes the hardware time duration between the starting and ending of notifiers.

Computes the hardware time duration between the starting notifier `start` and the ending notifier `end`. This API is used to measure the execution time of all the tasks between the starting notifier and ending notifier. The measuremen can be used to improve the performance.

Parameters

- `[in] start`: The handle of the starting notifier created by the *cnrtNotifierCreate* API.
- `[in] end`: The handle of ending notifier created by the *cnrtNotifierCreate* API.
- `[out] us`: The hardware time duration between the starting and ending of notifiers in microsecond.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorBusy*, *cnrtErrorCndrvFuncCall*

Note

- Call *cnrtPlaceNotifier* to place the starting and ending notifiers into the queue respectively first. Otherwise, *cnrtErrorCndrvFuncCall* is returned.
- If the *cnrtNotifierDuration* API is called before the starting notifier or ending notifier is placed into the queue, *cnrtErrorCndrvFuncCall* is returned.

Example

```
int main () {
  // Create notifier
  cnrtNotifier_t notifier_s;
  cnrtNotifier_t notifier_e;
  cnrtNotifierCreate(&notifier_s);
  cnrtNotifierCreate(&notifier_e);

  // Create queue
  cnrtQueue_t queue;
  cnrtQueueCreate(&queue);

  size_t size = 0x1000000;
  char *host_mem = (char *)malloc(size);
  void *dev_mem = NULL;
  cnrtMalloc(&dev_mem, size);

  // Place notifier into a queue
  cnrtPlaceNotifier(notifier_s, queue);
  // Push a task into queue between the two notifiers
  cnrtMemcpyAsync(dev_mem, host_mem, size, queue, cnrtMemcpyHostToDev);
  cnrtPlaceNotifier(notifier_e, queue);

  cnrtQueueSync(queue);

  // Query notifier
```

```
    cnrtQueryNotifier(notifier_s);
    cnrtQueryNotifier(notifier_e);

    // Wait for notifier
    cnrtWaitNotifier(notifier_s);
    cnrtWaitNotifier(notifier_e);

    // Compute the hardware duration
    float us;
    cnrtNotifierDuration(notifier_s, notifier_e, &us);
    printf("hardware time consuming between the two notifier is %f\n", us);

    cnrtNotifierDestroy(notifier_s);
    cnrtNotifierDestroy(notifier_e);

    cnrtQueueDestroy(queue);
    cnrtFree(dev_mem);
    free(host_mem);

    return 0;
}
```

4.5 Queue Management

4.5.1 cnrtQueueCreate

cnrtRet_t `cnrtQueueCreate`(*cnrtQueue_t* **pQueue*)

Creates a queue.

Creates a queue on the current device, and returns a pointer `pQueue` to the newly created queue. Define how the queues are synchronized with *cnrtSetDeviceFlag*. By default *cnrtDeviceScheduleSpin* is used. Call *cnrtGetDeviceFlag* to query the current behavior.

- *cnrtDeviceScheduleSpin*: CPU actively spins when waiting for the device execution result. For this option, the latency may be lower, but it may decrease the performance of CPU threads if the tasks are executed in parallel with MLU. This value is used by default.
- *cnrtDeviceScheduleBlock*: CPU thread is blocked on a synchronization primitive when waiting for the device execution result.
- *cnrtDeviceScheduleYield*: CPU thread yields when waiting for the device execution results.

For this option, the latency may be higher, but it can increase the performance of CPU threads if the tasks are executed in parallel with the device.

Parameters

- `[out] pQueue`: Pointer to the newly created queue.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- Call *cnrtQueueDestroy* to release the queue resources; otherwise, the memory leaks may occur.
- To learn more about queue lifecycle management, please see "Cambricon CNDrv Developer Guide".

Example

- See example in *cnrtQueueSync*.

4.5.2 cnrtQueueCreateWithPriority

```
cnrtRet_t cnrtQueueCreateWithPriority(cnrtQueue_t *pQueue,
                                      unsigned int flags,
                                      int priority)
```

Creates a queue with the specified priority.

Creates a queue on the current device with the specified priority, and returns a pointer `pQueue` to the newly created queue. If you want to define how the queues are synchronized with *cnrtSetDeviceFlag*, see description in *cnrtQueueCreate*.

Parameters

- `[out] pQueue`: Pointer to the newly created queue.
- `[in] flags`: Flag used in this operation, which is reserved for further use. It is recommended to set this parameter to 0.
- `[in] priority`: Priority of the queue. Lower numbers represent higher priorities. See *cnrtDeviceGetQueuePriorityRange* for more information about the meaningful queue priorities that can be passed.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- *cnrtDeviceGetQueuePriorityRange* can be called to query the range of meaningful numerical priorities. If the specified priority is out of the legal range returned by *cnrtDeviceGetQueuePriorityRange*, it will automatically be clamped to the lowest or the highest number in the legal range.

- Call *cnrtQueueDestroy* to release the queue resources; otherwise, the memory leaks may occur.

Example

```
int main () {
  void *dev_mem0 = NULL;
  void *dev_mem1 = NULL;
  cnrtMalloc(&dev_mem0, size);
  cnrtMalloc(&dev_mem1, size);

  int least_priority;
  int greatest_priority;

  cnrtDeviceGetQueuePriorityRange(&least_priority, &greatest_priority);

  // Create a queue
  cnrtQueue_t queue;
  cnrtQueueCreateWithPriority(&queue, 0, greatest_priority);

  int priority;
  cnrtQueueGetPriority(queue, &priority);
  printf("The priority of the queue is %d\n", priority);

  // Allocate memory on device and host
  size_t size = 0x1000000;
  char *host_mem0 = NULL;
  char *host_mem1 = NULL;
  host_mem0 = (char *)malloc(size);
  host_mem1 = (char *)malloc(size);

  // Copy data asynchronously in two queues
  cnrtMemcpyAsync(dev_mem0, host_mem0, size, queue, cnrtMemcpyHostToDev);
  cnrtMemcpyAsync(host_mem1, dev_mem1, size, queue, cnrtMemcpyDevToHost);

  // Query the status of a queue.
  printf("before sync queue, %d\n", cnrtQueueQuery(queue));
  cnrtQueueSync(queue);
  printf("after sync queue, %d\n", cnrtQueueQuery(queue));

  // Release resources
  cnrtQueueDestroy(queue);
  cnrtFree(dev_mem0);
  cnrtFree(dev_mem1);

  free(host_mem0);
  free(host_mem1);

  return 0;
}
```

4.5.3 cnrtQueueGetPriority

cnrtRet_t `cnrtQueueGetPriority`(*cnrtQueue_t* *queue*, int **priority*)

Queries the priority of a queue.

Queries the priority of a queue created using *cnrtQueueCreate* or *cnrtQueueCreateWithPriority* and returns the priority in `priority`.

Parameters

- `[in] queue`: The queue to query.
- `[out] priority`: Pointer to a signed integer in which the queue's priority is returned.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*, *cnrtErrorSysNoMem*

Note

- If the queue is created with a priority outside the numerical range returned by *cnrtDeviceGetQueuePriorityRange*, this API will return the clamped priority.

Example

- See example in *cnrtQueueSync*.

4.5.4 cnrtQueueDestroy

cnrtRet_t `cnrtQueueDestroy`(*cnrtQueue_t* *queue*)

Destroys a queue.

Destroys a `queue` that is created by *cnrtQueueCreate*. If the queue is still executing operations when *cnrtQueueDestroy* is called, this API will return immediately, but the resources associated with the queue are released automatically after all the operations in the queue are completed.

Parameters

- [in] queue: The queue to be destroyed.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- See example in *cnrtQueueSync*.

4.5.5 cnrtQueueQuery

cnrtRet_t `cnrtQueueQuery`(*cnrtQueue_t queue*)

Queries if a queue is completed.

Returns the status of the queue. If all the operations in the queue are completed, *cnrtSuccess* is returned. If the operations in the queue are still being executed, *cnrtErrorBusy* is returned.

Parameters

- [in] queue: The queue to query.

Return

- *cnrtSuccess*, *cnrtErrorNoCnrtContext*, *cnrtErrorBusy*, *cnrtErrorCndrvFuncCall*

Note

- If queue is set to NULL, the default queue will be used.

Example

```
int main () {
  // Create a queue
  cnrtQueue_t queue;
  cnrtQueueCreate(&queue);

  // Allocate memory on device and host
  size_t size = 0x1000000;
  char *host_mem0 = NULL;
  char *host_mem1 = NULL;
  host_mem0 = (char *)malloc(size);
  host_mem1 = (char *)malloc(size);
  void *dev_mem0 = NULL;
  void *dev_mem1 = NULL;
  cnrtMalloc(&dev_mem0, size);
  cnrtMalloc(&dev_mem1, size);  // Set flag, cause cnrtDeviceScheduleSpin is 0, so␣
↪that be equal.
// But before malloc is called, this operation is invalid.


    // Copy data asynchronously in two queues
    cnrtMemcpyAsync(dev_mem0, host_mem0, size, queue, cnrtMemcpyHostToDev);
    cnrtMemcpyAsync(host_mem1, dev_mem1, size, queue, cnrtMemcpyDevToHost);

    // Query the status of a queue.
    printf("before sync queue, %d\n", cnrtQueueQuery(queue));
    cnrtQueueSync(queue);
    printf("after sync queue, %d\n", cnrtQueueQuery(queue));

    // Release resources
    cnrtQueueDestroy(queue);

    cnrtFree(dev_mem0);
    cnrtFree(dev_mem1);

    free(host_mem0);
    free(host_mem1);

    return 0;
  }
```

4.5.6 cnrtQueueSync

cnrtRet_t `cnrtQueueSync`(*cnrtQueue_t queue*)

Waits for queue operations to be completed.

Blocks further executions in CPU thread until all the tasks in the queue on the current MLU device are completed.

Parameters

- `[in] queue`: The queue to be waited for.

Return

- *cnrtSuccess*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- If `queue` is set to NULL, the default queue will be used.
- This API may also return *cnrtErrorQueue* from previous and asynchronous operations.

Example

```
int main () {
  // Create a notifier
  cnrtNotifier_t notifier;
  cnrtNotifierCreate(&notifier);

  // Create a queue
  cnrtQueue_t queue_0;
  cnrtQueue_t queue_1;
  cnrtQueueCreate(&queue_0);
  cnrtQueueCreate(&queue_1);

  // Allocate memory on device and host
  size_t size = 0x100000;
  char *host_mem0 = NULL;
  char *host_mem1 = NULL;
  host_mem0 = (char *)malloc(size);
  host_mem1 = (char *)malloc(size);
  void *dev_mem0 = NULL;
  void *dev_mem1 = NULL;
  cnrtMalloc(&dev_mem0, size);
  cnrtMalloc(&dev_mem1, size);

  // Copy data asynchronously in two queues
  cnrtMemcpyAsync(dev_mem0, host_mem0, size, queue_0, cnrtMemcpyHostToDev);
  cnrtMemcpyAsync(host_mem1, dev_mem1, size, queue_1, cnrtMemcpyDevToHost);
  // Put a notifier into a queue
  cnrtPlaceNotifier(notifier, queue_0);

  // Synchronize two queues
  cnrtQueueWaitNotifier(notifier, queue_1, 0);

  // Wait until all tasks are completed
  cnrtQueueSync(queue_0);
  cnrtQueueSync(queue_1);

  // Release resources
  cnrtQueueDestroy(queue_0);
  cnrtQueueDestroy(queue_1);
  cnrtNotifierDestroy(notifier);

  cnrtFree(dev_mem0);
  cnrtFree(dev_mem1);
  free(host_mem0);
  free(host_mem1);

  return 0;
}
```

4.5.7 cnrtQueueWaitNotifier

```
cnrtRet_t cnrtQueueWaitNotifier(cnrtNotifier_t notifier,
                                cnrtQueue_t queue,
                                unsigned int flag)
```

Waits all the preceding tasks before a notifier to be completed before the specified queue does any further executions.

Blocks all future tasks in the queue until all the preceding tasks before a notifier are completed. The queue only waits for the completion of the most recent host call of *cnrtPlaceNotifier* on the notifier. This API is used to synchronize the queue efficiently on the device.

Parameters

- [in] notifier: The notifier to be waited for.
- [in] queue: The queue to wait.
- [in] flag: Flag used in this operation, which is reserved for further use. It is recommended to set this parameter to 0.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorNoCnrtContext*, *cnrtErrorCndrvFuncCall*

Note

- If queueis set to NULL, the default queue will be used.
- This API is used for synchronization across queues. To synchronize in a single queue, use *cnrtWaitNotifier*.

Example

- See example in *cnrtQueueSync*.

4.5.8 cnrtQueueSetAttribute

```
cnrtRet_t cnrtQueueSetAttribute(cnrtQueue_t queue,
                                cnrtQueueAttrID_t attr_id,
                                const cnrtQueueAttrValue_t *value)
```

Sets the queue attribute.

Sets the attributes corresponding to `attr_id` for `queue` from corresponding attribute of `value`.

Parameters

- `[in] queue`: The queue handle to be set.
- `[in] attr_id`: The attribute ID.
- `[in] value`: The attribute value to set.

Return

- *cnrtSuccess*, *cnrtErrorNotSupport*, *cnrtErrorArgsInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- None.

4.5.9 cnrtQueueGetAttribute

```
cnrtRet_t cnrtQueueGetAttribute(cnrtQueue_t queue,
                                cnrtQueueAttrID_t attr_id,
                                cnrtQueueAttrValue_t *value_out)
```

Queries the queue attribute.

Queries the attributes corresponding to `attr_id` for `queue`, and stores it in corresponding member of `value`.

Parameters

- `[in] queue`: The queue handle to query.
- `[in] attr_id`: The attribute ID.
- `[out] value_out`: The room to store the attribute value.

Return

- *cnrtSuccess*, *cnrtErrorNotSupport*, *cnrtErrorArgsInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- None.

4.5.10 cnrtQueueCopyAttributes

```
cnrtRet_t cnrtQueueCopyAttributes(cnrtQueue_t dst,
                                  cnrtQueue_t src)
```

Copies queue attributes from source queue to destination queue.

Copies attributes from source queue `src` to destination queue `dst`.

Parameters

- `[in] dst`: The destination queue.
- `[in] src`: The source queue.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

- None.

4.5.11 cnrtQueueBeginCapture

```
cnrtRet_t cnrtQueueBeginCapture(cnrtQueue_t queue,
                                enum cnrtQueueCaptureMode mode)
```

Begins capture on a queue.

When the `queue` is capturing, all the tasks pushed to this queue will not be executed but instead captured into a Task Topo. Call *cnrtQueueIsCapturing()* to query whether the queue is capturing. Call *cnrtQueueGetCaptureInfo()* to query the unique ID representing the sequence number of the capturing and other information.

Parameters

- `[in] queue`: The queue to begin capture for.
- `[in] mode`: The capture mode.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorIllegalState*

Note

- If `mode` is not *cnrtQueueCaptureModeRelaxed*, *cnrtQueueEndCapture()* must be called on this `queue` from the same thread.
- It is not allowed to pass to `queue` with NULL as a default queue to begin capturing.
- The capture must be ended on the same queue as it is started.

Example

```
int main() {
  cnrtTaskTopo_t topo;
  cnrtTaskTopoEntity_t entity;
  uint64_t id; // the capture sequence unique id.
  const cnrtTaskTopoNode_t *dependencies;
  size_t numDependencies;
  cnrtQueue_t queue;
  cnrtQueueCaptureStatus_t status;
  cnrtTaskTopoNode_t node0, node1;
  cnrtTaskTopoNodeType_t type;

  cnrtRet_t ret = cnrtQueueCreate(&queue);
  if (ret != cnrtSuccess) return ret;

  ret = cnrtQueueBeginCapture(queue, cnrtQueueCaptureModeRelaxed);
  if (ret != cnrtSuccess) return ret;

  ret = cnrtQueueIsCapturing(queue, &status);
  if (ret != cnrtSuccess || status != cnrtQueueCaptureStatusActive) return ret;

  // Capture the kernel task.
  cnrtDim3_t dim = {1, 1, 1};
  kernel<<<dim, cnrtFuncTypeBlock, queue>>>();

  ret = cnrtQueueGetCaptureInfo(queue, &status, &id, &topo,
                                &dependencies, &numDependencies);
  if (ret != cnrtSuccess) return ret;
  assert(numDependencies == 1);
  node0 = dependencies[0];
  ret = cnrtTaskTopoNodeGetType(dependencies[0], type);
  assert(ret == cnrtSuccess);
  assert(cnrtTaskTopoNodeTypeKernel == type);

  // Capture the host function task.
  cnrtInvokeHostFunc(queue, [](void *args) {//...}, nullptr);

  ret = cnrtQueueGetCaptureInfo(queue, &status, &id, &topo,
                                &dependencies, &numDependencies);
  if (ret != cnrtSuccess) return ret;

  // Nodes in topo are {kernel_node -> host_node}
  node1 = dependencies[0];
  assert(status == cnrtQueueCaptureStatusActive);
  assert(numDependencies == 1);
  ret = cnrtTaskTopoNodeGetType(node1, &type);
  assert(ret == cnrtSuccess);
  assert(type == cnrtTaskTopoNodeTypeHost);

  cnrtTaskTopoNode_t updated_deps[2] = {node0, node1};
  ret = cnrtQueueUpdateCaptureDependencies(queue, updated_deps, 2,
                                           cnrtQueueSetCaptureDependencies);
  if (ret != cnrtSuccess) return ret;

  ret = cnrtQueueGetCaptureInfo(queue, &status, &id, &topo,
                                &dependencies, &numDependencies);
  if (ret != cnrtSuccess) return ret;
  assert(numDependencies == 2);
  assert((dependencies[0] == node0 && dependencies[1] == node1) ||
         (dependencies[0] == node1 && dependencies[1] == node0));

  // Invoke more asynchronous tasks
  // ...

  ret = cnrtQueueEndCapture(queue, &topo);
  if (ret != cnrtSuccess) return ret;

  ret = cnrtTaskTopoInstantiate(&entity, topo, nullptr, nullptr, 0);
  if (ret != cnrtSuccess) return ret;

  // Invoke TaskTopo, real time to invoke tasks
  ret = cnrtTaskTopoEntityInvoke(entity, queue);
  if (ret != cnrtSuccess) return ret;

  // ...
}
```

4.5.12 cnrtQueueEndCapture

```
cnrtRet_t cnrtQueueEndCapture(cnrtQueue_t queue,
                              cnrtTaskTopo_t *pTaskTopo)
```

Ends a queue capture, and returns the captured Task Topo.

Ends a queue capture sequence that is begun with *cnrtQueueBeginCapture()* and returns the captured Task Topo in `pTaskTopo`. If the queue is not in capture status, the API call returns *cnrtErrorIllegalState*.

Parameters

- `[in]` `queue`: The queue in which to end capturing.
- `[out]` `pTaskTopo`: The captured Task Topo.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorIllegalState*, *cnrtErrorQueueCaptureInvalidated*

Note

- If the queue is not capturing under *cnrtQueueCaptureModeRelaxed*, *cnrtQueueEndCapture* must be called on this `queue` from the same thread.
- If the capture is invalidated, then the NULL Task Topo will be returned.
- Capture must have been begun on `queue` via *cnrtQueueBeginCapture*.

Example

- See example in *cnrtQueueBeginCapture()*.

4.5.13 cnrtQueueIsCapturing

```
cnrtRet_t cnrtQueueIsCapturing(cnrtQueue_t queue,
                               enum cnrtQueueCaptureStatus *pStatus)
```

Queries a queue's capture status.

Returns the `queue` capture status via `pStatus`. If the `queue` is not in capture status, *cnrtQueueCaptureStatusNone* is returned. If the `queue` is in capture status but the capture sequence has been invalidated due to previous error, *cnrtQueueCaptureStatusInvalidated* is returned; otherwise *cnrtQueueCaptureStatusActive* is returned.

Parameters

- `[in]` `queue`: The queue to query.
- `[out]` `pStatus`: The capture status.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- See example in *cnrtQueueBeginCapture()*.

4.5.14 cnrtQueueGetCaptureInfo

```
cnrtRet_t cnrtQueueGetCaptureInfo(cnrtQueue_t queue,
                                  enum cnrtQueueCaptureStatus *pStatus,
                                  uint64_t *idOut,
                                  cnrtTaskTopo_t *pTaskTopo,
                                  const cnrtTaskTopoNode_t **pDependenciesOut,
                                  size_t *pNumDependencies)
```

Queries a queue's capture information.

Returns detailed information if the `queue` is in active capture status. The parameters `idOut`, `pTaskTopo`, `pDependenciesOut`, `pNumDependencies` are optional, which can be NULL, and nothing is returned.

Parameters

- `[in] queue`: The queue to query.
- `[out] pStatus`: The capture status.
- `[out] idOut`: The unique sequence number of current capturing.
- `[out] pTaskTopo`: The current captured Task Topo.
- `[out] pDependenciesOut`: A pointer to store an array of dependency nodes.
- `[out] pNumDependencies`: The count of dependency nodes.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- See example in *cnrtQueueBeginCapture()*.

4.5.15 cnrtQueueUpdateCaptureDependencies

cnrtRet_t `cnrtQueueUpdateCaptureDependencies`(*cnrtQueue_t* *queue*, *cnrtTaskTopoNode_t* **dependencies*, size_t *numDependencies*, unsigned int *flags*)

Updates the set of dependencies in a capturing queue.

Modifies the dependency set of capturing `queue`. The dependency set is the set of nodes that the next captured node in the `queue` will depend on.

Parameters

- `[in] queue`: The queue in capture status.
- `[in] dependencies`: The array of dependency nodes to modify the capturing sequence dependencies.
- `[in] numDependencies`: The node count of the `dependencies`.
- `[in] flags`: Modification flag. Valid flags are *cnrtQueueAddCaptureDependencies* and *cnrtQueueSetCaptureDependencies*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorIllegalState*, *cnrtErrorQueueCaptureInvalidated*, *cnrtErrorSysNoMem*

Note

- None.

Example

- See example in *cnrtQueueBeginCapture()*.

4.6 Version Management

4.6.1 cnrtDriverGetVersion

cnrtRet_t `cnrtDriverGetVersion`(int **major*, int **minor*, int **patch*)

Retrieves the version of the current driver.

Returns the major version in `major`, minor version in `minor`, and patch version in `patch` of the current driver.

Parameters

- [out] major: Pointer to the major of version.
- [out] minor: Pointer to the minor of version.
- [out] patch: Pointer to the patch of version.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorCndrvFuncCall*

Note

- None.

Example

```
int main () {
  int major, minor, patch;
  cnrtDriverGetVersion(&major, &minor, &patch);
  printf("driver version is %d.%d.%d\n", major, minor, patch);

  return 0;
}
```

4.6.2 cnrtGetLibVersion

cnrtRet_t cnrtGetLibVersion(int **major*,
int **minor*,
int **patch*)

Retrieves the version of the current CNRT.

Returns the major version in major, minor version in minor, and patch version in patch of the current CNRT instance.

Parameters

- [out] major: Pointer to the major of version.
- [out] minor: Pointer to the minor of version.
- [out] patch: Pointer to the patch of version.

Return

- *cnrtSuccess*

Note

- None.

Example

```
int main () {
  int major, minor, patch;
  cnrtGetLibVersion(&major, &minor, &patch);
  printf("cnrt version is %d.%d.%d\n", major, minor, patch);

  return 0;
}
```

4.7 Error Handling Management

4.7.1 cnrtGetErrorName

```
const char *cnrtGetErrorName(cnrtRet_t error)
```

Retrieves the error name of an error code.

Returns the string containing the name of an error code in the enum.

Parameters

- `[in] error`: The error code to convert to string.

Return

- A pointer to string of the error code.

Note

- None.

Example

- See example in *cnrtGetLastError* API.

4.7.2 cnrtGetErrorStr

```
const char *cnrtGetErrorStr(cnrtRet_t error)
```

Retrieves the error message of an error code.

Returns the description string for an error code.

Parameters

- `[in] error`: The error code to convert to string.

Return

- A pointer to string message according to the error code.

Note

- None.

Example

- See example in *cnrtGetLastError* API.

4.7.3 cnrtGetLastError

cnrtRet_t `cnrtGetLastError`(void)

Retrieves the last error from CNRT API call.

Returns the last error code returned from the CNRT API call in the same host thread.

Return

- *cnrtSuccess*, *cnrtErrorNotReady*, *cnrtErrorNoDevice*, *cnrtErrorDeviceInvalid*, *cnrtErrorArgsInvalid*, *cnrtErrorSys*, *cnrtErrorSysNoMem*, *cnrtErrorInvalidResourceHandle*, *cnrtErrorIllegalState*, *cnrtErrorNotSupport*, *cnrtErrorOpsNotPermitted*, *cnrtErrorQueue*, *cnrtErrorNoMem*, *cnrtErrorAssert*, *cnrtErrorKernelTrap*, *cnrtErrorKernelUserTrap*, *cnrtErrorNotFound*, *cnrtErrorInvalidKernel*, *cnrtErrorNoKernel*, *cnrtErrorNoModule*, *cnrtErrorQueueCaptureUnsupported*, *cnrtErrorQueueCaptureInvalidated*, *cnrtErrorQueueCaptureWrongThread*, *cnrtErrorQueueCaptureMerged*, *cnrtErrorQueueCaptureUnjoined*, *cnrtErrorQueueCaptureIsolation*, *cnrtErrorQueueCaptureUnmatched*, *cnrtErrorTaskTopoEntityUpdateFailure*, *cnrtErrorSetOnActiveProcess*, *cnrtErrorDevice*, *cnrtErrorNoAttribute*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorCndrvFuncCall*, *cnrtErrorNoCnrtContext*, *cnrtErrorInvalidSymbol*

Note

- Once *cnrtGetLastError* is called, the error code will be reset to *cnrtSuccess*.

Example

```
// __mlu_global__ function must be written in a single file named *.mlu
__mlu_global__ void bangKernelAdd(int *a, int *b, unsigned int size) {
  for (int i = 0; i < size; i++) {
    b[i] += a[i];
  }
}
void hostAdd(int *x, int *y, unsigned int size,
              cnrtDim3_t dim, cnrtFunctionType_t type, cnrtQueue_t queue) {
  bangKernelAdd<<<dim, type, queue>>>(x, y, size);
}

// Hybrid programming
int main () {
  int a, b;
  uint32_t seed = 0x123;
  a = (int)rand_r(&seed) % 10000;
  b = (int)rand_r(&seed) % 10000;
  cnrtDim3_t dim;
  dim.x = 1;
  dim.y = 1;
  dim.z = 1;
  cnrtFunctionType_t type = cnrtFuncTypeBlock;

  // Ensure the origin host thread error code is cnrtSuccess.
  cnrtGetLastError();
  cnrtQueue_t queue;
  cnrtQueueCreate(&queue);

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

  // Launch kernel
  hostAdd(k_a, k_b, 1, dim, type, queue);
  cnrtRet_t ret = cnrtPeekAtLastError();
  printf("cnrtGetErrorName: %s\n", cnrtGetErrorName(ret));
  printf("cnrtGetErrorStr: %s\n", cnrtGetErrorStr(cnrtGetLastError()));
  cnrtSyncDevice();

  // Free resource
  cnrtFree(k_a);
  cnrtFree(k_b);
  cnrtQueueDestroy(queue);

  return 0;
}
```

4.7.4 cnrtPeekAtLastError

cnrtRet_t `cnrtPeekAtLastError`(void)

Retrieves the last error from the CNRT API call without resetting.

Returns the last error code returned from the CNRT API call.

Return

- *cnrtSuccess*, *cnrtErrorNotReady*, *cnrtErrorNoDevice*, *cnrtErrorDeviceInvalid*, *cnrtErro-*

rArgsInvalid, *cnrtErrorSys*, *cnrtErrorSysNoMem*, *cnrtErrorInvalidResourceHandle*, *cnrtErrorIllegalState*, *cnrtErrorNotSupport*, *cnrtErrorOpsNotPermitted*, *cnrtErrorQueue*, *cnrtErrorNoMem*, *cnrtErrorAssert*, *cnrtErrorKernelTrap*, *cnrtErrorKernelUserTrap*, *cnrtErrorNotFound*, *cnrtErrorInvalidKernel*, *cnrtErrorNoKernel*, *cnrtErrorNoModule*, *cnrtErrorQueueCaptureUnsupported*, *cnrtErrorQueueCaptureInvalidated*, *cnrtErrorQueueCaptureWrongThread*, *cnrtErrorQueueCaptureMerged*, *cnrtErrorQueueCaptureUnjoined*, *cnrtErrorQueueCaptureIsolation*, *cnrtErrorQueueCaptureUnmatched*, *cnrtErrorTaskTopoEntityUpdateFailure*, *cnrtErrorSetOnActiveProcess*, *cnrtErrorDevice*, *cnrtErrorNoAttribute*, *cnrtErrorMemcpyDirectionInvalid*, *cnrtErrorCndrvFuncCall*, *cnrtErrorNoCnrtContext*, *cnrtErrorInvalidSymbol*

Note

- The error code will not be reset to *cnrtSuccess* after calling *cnrtPeekAtLastError*.

Example

- See example in *cnrtGetLastError*.

4.8 Utility Management

4.8.1 cnrtCreateQuantizedParam

cnrtRet_t `cnrtCreateQuantizedParam(`*cnrtQuantizedParam_t* **param*,
int *pos*,
float *scale*,
int *offset*)

Creates and sets the quantization parameters used for casting data types.

Creates the quantization parameters `pos`, `scale`, and `offset`, and sets the values to these parameters. Returns in `param` the quantization parameters used for casting data types. For more information about quantization, see "Cambricon BANG C/C++ Programming Guide".

Parameters

- `[out] param`: Pointer to the quantization parameters defined in *cnrtQuantizedParam_t*.
- `[in] pos`: The position factor used in quantization.
- `[in] scale`: The scale factor used in quantization.
- `[in] offset`: The offset factor used in quantization.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- After this API is called, the output pointer `param` is used in the *cnrtCastDataType* or cnrtTransOrderAndCast API to cast data type.
- The *cnrtDestroyQuantizedParam* API should be called to release the memory space when `param` is no longer needed.

4.8.2 cnrtCreateQuantizedParamByChannel

```
cnrtRet_t cnrtCreateQuantizedParamByChannel(cnrtQuantizedParam_t *param,
                                            int *poses,
                                            float *scales,
                                            float *offsets,
                                            int dimNum,
                                            int *dimValues,
                                            int channelDim)
```

Creates and sets the quantization parameters used for casting data types. Quantizes data by channel.

Creates quantization parameters `poses`, `scales` and `offsets`, and sets the values to these parameters. Returns in `param` the quantization parameters defined in *cnrtQuantizedParam_t* used for casting data type. The data is divided into groups based on the number of elements of the channel dimension and quantized filter data for each group. The *cnrtCreateQuantizedParam* API quantizes data without division, and is usually for input or output data. Compared with the *cnrtCreateQuantizedParam* API, this API has a higher precision quantization. For more information about quantization, see "Cambricon BANG C/C++ Programming Guide".

Parameters

- `[out] param`: Pointer to the quantization parameters defined in *cnrtQuantizedParam_t*.
- `[in] poses`: The position factor used in quantization.
- `[in] scales`: The scale factor used in quantization.
- `[in] offsets`: The offset factor used in quantization.
- `[in] dimNum`: The number of dimensions of the filter data to be quantized.
- `[in] dimValues`: The number of elements for each dimension of the filter data to be quantized.
- `[in] channelDim`: The dimension index of the channel in the filter data layout.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSys*

Note

- After this API is called, the output pointer `param` is used in the *cnrtCastDataType* or cnrtTransOrderAndCast API to cast data type.

- The *cnrtDestroyQuantizedParam* API should be called to release the memory space when `param` is no longer needed.
- This API is to set parameters for quantization by channel, so the number of `scales`, `poses`, `offsets` should be the same as that of channel elements.

4.8.3 cnrtDestroyQuantizedParam

cnrtRet_t **cnrtDestroyQuantizedParam**(*cnrtQuantizedParam_t param*)

Releases the memory resources of the quantization parameters.

Destroys the quantization parameters `param` and cleans up the parameter resources.

Parameters

- `[in] param`: Pointer to the quantization parameters defined in *cnrtQuantizedParam_t*.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSys*

Note

- None.

4.8.4 cnrtCastDataType

cnrtRet_t **cnrtCastDataType**(void **src*,
cnrtDataType_t srcDataType,
void **dst*,
cnrtDataType_t dstDataType,
int *count*,
cnrtQuantizedParam_t param)

Converts the data into another data type.

Converts data pointed by `src` in `srcDataType` data type into the `dstDataType` data type with data quantization if the quantization parameter `param` is not set to NULL. Returns the converted data in `dst`.

Parameters

- `[out] dst`: Pointer to the converted output data.
- `[in] src`: Pointer to the input data to be converted.
- `[in] srcDataType`: The data type of the input data to be converted. The data type is defined in *cnrtDataType_t*.
- `[in] dstDataType`: The data type of the data to be converted. The data type is defined in *cnrtDataType_t*.

- [in] count: The number of data to be converted.
- [in] param: Pointer to the quantization parameters defined in *cnrtQuantizedParam_t*. Create and set quantization parameters via the *cnrtCreateQuantizedParam* or *cnrtCreateQuantizedParamByChannel* API. To ignore quantizing data, set this parameter to NULL.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- If the input pointer param is set to NULL, the data will not be quantized. The data types are shown in the order srcDataType-dstDataType.
  Supported combinations: float32-float16, float32-uint8, int64-float16, float16-float32, float16-uint8, uint8-float32, uint8-float16, float32-float32 in int64-float16 case, int64 is first converted to float,and then the float data is converted.
- If the input pointer param is not set to NULL, the data will be quantized. The data types are shown in the order srcDataType-dstDataType.
  Supported combinations: float32-float16, float32-int16, float32-int8, float32-int32, int32-float32, float16-int16, int16-float32, int8-float32, float32-float32

4.8.5 cnrtCastDataType_V2

```
cnrtRet_t cnrtCastDataType_V2(const void *src,
                              cnrtDataType_V2_t srcDataType,
                              void *dst,
                              cnrtDataType_V2_t dstDataType,
                              int count,
                              cnrtQuantizedParam_t param,
                              cnrtRoundingMode_t roundingMode)
```

Converts the input data into another data type in specified rounding mode.

Converts data pointed by src in srcDataType data type into the dstDataType data type with data quantization if the quantization parameter param is not set to NULL. Returns the converted data in dst.

Parameters

- [out] dst: Pointer to the converted output data.
- [in] src: Pointer to the input data to be converted.
- [in] srcDataType: The data type of the input data to be converted. The data type is defined in *cnrtDataType_t*.
- [in] dstDataType: The data type of the data to be converted. The data type is defined

in *cnrtDataType_t*.

- `[in] count`: The number of data to be converted.
- `[in] param`: Pointer to the quantization parameters defined in *cnrtQuantizedParam_t*. Create and set quantization parameters via the *cnrtCreateQuantizedParam* or *cnrtCreateQuantizedParamByChannel* API. To ignore quantizing data, set this parameter to NULL.
- `[in] roundingMode`: The rounding mode for data type conversion.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- If the input value `count` is zero, the API will return `cnrtSuccess` .
- If the input pointer `param` is set to NULL, the data will not be quantized. The data types are shown in the order `srcDataType-dstDataType`.

  Supported combinations: double->half, double->bfloat, float->half, float->bfloat16, float->uint64, float->int64, float->uint32, float->int32, float->uint16, float->int16, float->uint8, float->int8, float->bool, bfloat->float, bfloat->uint64, bfloat->uint32, bfloat->uint16, bfloat->int64, bfloat->int32, half->float, half->bfloat, half->uint64, half->int64, half->uint32, half->int32, half->uint16, half->int16, half->uint8, half->int8, uint64->half, uint64->bfloat, uint64->float, int64->half, int64->bfloat, int64->float, uint32->half, uint32->bfloat, uint32->float, int32->half, int32->bfloat, int32->float, uint16->half, uint16->bfloat, uint16->float, int16->half, int16->bfloat, int16->float, uint8->half, uint8->bfloat, uint8->float, int8->half, int8->bfloat, int8->float.

- If the input pointer `param` is not set to NULL, the data will be quantized. The data types are shown in the order `srcDataType-dstDataType`.

  Supported combinations: float->half, float->int16, float->int8, float->int32, int32->float, half->int16, half->int8, int16->float, int16->half, int8->float, int8->half.

4.8.6 cnrtFilterReshape

cnrtRet_t **`cnrtFilterReshape`**(void **dst*,
void **src*,
int *n*,
int *h*,
int *w*,
int *c*,
cnrtDataType_t *type*)

Reshapes the input filter data.

Reshapes the filter data from source memory address pointed by `dst` to destination memory address pointed by `src` with the source data shape src[NHWC] and data type `type` that is specified in `cnrtDataType_t` enum.

Parameters

- `[out] dst`: The destination address.
- `[in] src`: The source address.
- `[in] n`: The batch size.
- `[in] h`: The height.
- `[in] w`: The width.
- `[in] c`: The channel.
- `[in] type`: The data type of the source and destination data.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*,

Note

- None.

Example

- None.

4.9 Task Topo Management

4.9.1 cnrtTaskTopoCreate

cnrtRet_t `cnrtTaskTopoCreate`(*cnrtTaskTopo_t* **pTaskTopo*, unsigned int *flags*)

Creates a Task Topo.

Creates a new Task Topo, and returns a pointer `pTaskTopo` to the newly created Task Topo.

Parameters

- `[out] pTaskTopo`: Pointer to the newly created Task Topo.
- `[in] flags`: The Task Topo creation flags, which must be 0.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

```
int main () {
  cnrtTaskTopo_t topo;
  cnrtRet_t res = cnrtTaskTopoCreate(&topo, 0);
  assert(res == cnrtSuccess);
}
```

4.9.2 cnrtTaskTopoDestroy

cnrtRet_t `cnrtTaskTopoDestroy`(*cnrtTaskTopo_t taskTopo*)

Destroys a Task Topo.

Destroys the Task Topo specified by `taskTopo` and cleans up all of its nodes.

Parameters

- `[in] taskTopo`: The Task Topo to be destroyed.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.3 cnrtTaskTopoClone

cnrtRet_t `cnrtTaskTopoClone`(*cnrtTaskTopo_t *pTaskTopoClone*, *cnrtTaskTopo_t originalTaskTopo*)

Clones a Task Topo.

Creates a copy of `originalTaskTopo` and returns it in `pTaskTopo`. All parameters are copied into the cloned Task Topo.

Parameters

- `[out] pTaskTopoClone`: Pointer to the newly created cloned Task Topo.
- `[in] originalTaskTopo`: The Task Topo to clone.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.4 cnrtTaskTopoNodeFindInClone

```
cnrtRet_t cnrtTaskTopoNodeFindInClone(cnrtTaskTopoNode_t *pNode,
                                      cnrtTaskTopoNode_t originalNode,
                                      cnrtTaskTopo_t clonedTaskTopo)
```

Finds the corresponding node in cloned Task Topo.

Returns the node in `clonedTaskTopo` corresponding to `originalNode` in the original Task Topo.

Parameters

- `[out] pNode`: Pointer to the cloned node.
- `[in] originalNode`: The original node.
- `[in] clonedTaskTopo`: The cloned Task Topo to query.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.5 cnrtTaskTopoDestroyNode

```
cnrtRet_t cnrtTaskTopoDestroyNode(cnrtTaskTopoNode_t node)
```

Removes a node from the Task Topo.

Removes `node` from its Task Topo. This operation also severs dependencies of other nodes on `node` and vice versa.

Parameters

- `[in] node`: Node to be removed.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.6 cnrtTaskTopoGetEdges

```
cnrtRet_t cnrtTaskTopoGetEdges(cnrtTaskTopo_t taskTopo,
                               cnrtTaskTopoNode_t *from,
                               cnrtTaskTopoNode_t *to,
                               size_t *numEdges)
```

Returns dependency edges of a Task Topo.

Returns a list of dependency edges of the Task Topo. Edges are returned via corresponding indices in `from` and `to`, the node in `to[i]` has a dependency on the node in `from[i]`. `from` and `to` may both be NULL, in which case this API only returns the number of edges in `numEdges`, otherwise, `numEdges` entries will be filled in. If `numEdges` is higher than the actual number of edges, the remaining entries in `from` and `to` will be set to NULL, and the number of edges actually returned will be written to `numEdges`.

Parameters

- `[in] taskTopo`: Task Topo to get the edges from.
- `[out] from`: Location to return the edge of a source node.
- `[out] to`: Location to return the edge of a destination node.
- `[inout] numEdges`: See the description above for details.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- `from` and `to` must be both NULL or neither NULL, otherwise *cnrtErrorArgsInvalid* will be returned.

Example

- None.

4.9.7 cnrtTaskTopoGetNodes

```
cnrtRet_t cnrtTaskTopoGetNodes(cnrtTaskTopo_t taskTopo,
                               cnrtTaskTopoNode_t *pNode,
                               size_t *numNodes)
```

Returns nodes of a Task Topo.

Returns a list of nodes of a Task Topo. `pNode` may be NULL, in which case this API will return the number of nodes in `numNodes`. Otherwise, `numNodes` entries will be filled in. If `numNodes` is higher than the actual number of nodes, the remaining entries in `pNode` will be set to NULL, and the number of nodes actually obtained will be returned in `numNodes`.

Parameters

- `[in] taskTopo`: Task Topo to query.
- `[out] pNode`: The array to store the returned nodes.
- `[inout] numNodes`: See the description above for details.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.8 cnrtTaskTopoGetRootNodes

cnrtRet_t `cnrtTaskTopoGetRootNodes`(*cnrtTaskTopo_t* *taskTopo*, *cnrtTaskTopoNode_t* **pRootNode*, size_t **numRootNodes*)

Returns root nodes of a Task Topo.

Returns a list of root nodes of a Task Topo. `pRootNode` may be NULL, in which case this API will return the number of nodes in `numRootNodes`. Otherwise, `numRootNodes` entries will be filled in. If `numRootNodes` is higher than the actual number of nodes, the remaining entries in `pRootNode` will be set to NULL, and the number of nodes actually obtained will be returned in `pRootNode`.

Parameters

- `[in] taskTopo`: Task Topo to query.
- `[out] pRootNode`: The array to store the returned root nodes.
- `[inout] numRootNodes`: See the description above for details.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.9 cnrtTaskTopoAddDependencies

```
cnrtRet_t cnrtTaskTopoAddDependencies(cnrtTaskTopo_t taskTopo,
                                      const cnrtTaskTopoNode_t *from,
                                      const cnrtTaskTopoNode_t *to,
                                      size_t numDependencies)
```

Adds dependency edges to a Task Topo.

The number of dependencies to be added is defined by `numDependencies`. Elements in `from` and `to` at corresponding indices define a dependency. Each node in `from` and `to` must belong to `taskTopo`.

Parameters

- `[in] taskTopo`: The Task Topo to add dependency edges to.
- `[in] from`: Array of nodes that provide the dependencies.
- `[in] to`: Array of dependent nodes.
- `[in] numDependencies`: The number of dependencies to be added.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.10 cnrtTaskTopoRemoveDependencies

```
cnrtRet_t cnrtTaskTopoRemoveDependencies(cnrtTaskTopo_t taskTopo,
                                         const cnrtTaskTopoNode_t *from,
                                         const cnrtTaskTopoNode_t *to,
                                         size_t numDependencies)
```

Removes dependency edges from a Task Topo.

The number of dependencies to be removed is defined by `numDependencies`. Elements in `from` and `to` at corresponding indices define a dependency. Each node in `from` and `to` must belong to `taskTopo`

Parameters

- `[in] taskTopo`: The Task Topo to remove dependency edges from.
- `[in] from`: Array of nodes that provide the dependencies.
- `[in] to`: Array of dependent nodes.

- [in] numDependencies: The number of dependencies to be removed.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.11 cnrtTaskTopoNodeGetDependencies

cnrtRet_t cnrtTaskTopoNodeGetDependencies(*cnrtTaskTopoNode_t* *node*, *cnrtTaskTopoNode_t* **pDependencies*, size_t **numDependencies*)

Returns a node's dependencies.

Returns a list of dependencies of node. pDependencies may be NULL, in which case this API will return the number of dependencies in numDependencies. Otherwise, numDependencies entries will be filled in. If numDependencies is higher than the actual number of dependencies, the remaining entries in pDependencies will be set to NULL, and the number of nodes actually obtained will be returned in numDependencies.

Parameters

- [in] node: The node to query.
- [out] pDependencies: The array to store returned dependency of a node.
- [inout] numDependencies: See the description above for details.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.12 cnrtTaskTopoNodeGetDependentNodes

cnrtRet_t cnrtTaskTopoNodeGetDependentNodes(*cnrtTaskTopoNode_t* *node*, *cnrtTaskTopoNode_t* **pDependentNodes*, size_t **numDependentNodes*)

Returns a node's dependent nodes.

Returns a list of dependent nodes of `node`. `pDependentNodes` may be NULL, in which case this API will return the number of dependent nodes in `numDependentNodes`. Otherwise, `numDependentNodes` entries will be filled in. If `numDependentNodes` is higher than the actual number of dependent nodes, the remaining entries in `pDependentNodes` will be set to NULL, and the number of nodes actually obtained will be returned in `pDependentNodes`.

Parameters

- `[in] node`: The node to query.
- `[out] pDependentNodes`: The array to store returned dependent nodes of a node.
- `[inout] numDependentNodes`: See the description above for details.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.13 cnrtUserObjectCreate

cnrtRet_t `cnrtUserObjectCreate`(*cnrtUserObject_t* * *object_out*,
void * *ptr*,
cnrtHostFn_t *destroy*,
unsigned int *initialRefcount*,
unsigned int *flags*)

Creates a user object.

Creates a user object with the specified destructor callback and initial reference count. The initial references are owned by the caller. Destructor callbacks cannot make CNRT or CNDrv API calls and should avoid blocking behavior.

Parameters

- `[out] object_out`: Location to return the user object handle.
- `[in] ptr`: Pointer to pass the destroy function.
- `[in] destroy`: Callback to free the user object when it is no longer in use.
- `[in] initialRefcount`: The initial reference count to create the object with, which is typically 1.
- `[in] flags`: Currently it is required to pass 1. This indicates that the destroy callback cannot be waited on by any CNRT or CNDrv API. If you require synchronization of the callback, you should signal its completion manually.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.14 cnrtUserObjectAcquire

```
cnrtRet_t cnrtUserObjectAcquire(cnrtUserObject_t object,
                                unsigned int count)
```

Acquires a reference for a user object.

Acquires new references for a user object. The new references are owned by the caller.

Parameters

- [in] object: The user object to acquire a reference for.
- [in] count: The number of reference to acquire, which is typically 1. The value must be nonzero and not larger than INT_MAX.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.15 cnrtUserObjectRelease

```
cnrtRet_t cnrtUserObjectRelease(cnrtUserObject_t object,
                                unsigned int count)
```

Releases a reference for a user object.

Releases user object references owned by the caller. The user object's destructor is invoked if the reference count reaches zero. It is undefined behavior to release references not owned by the caller, or to use object handle after all references are released.

Parameters

- [in] object: The user object to release a reference for.
- [in] count: The number of reference to release, which is typically 1. The value must be nonzero and not larger than INT_MAX.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.16 cnrtTaskTopoAcquireUserObject

cnrtRet_t `cnrtTaskTopoAcquireUserObject(`*cnrtTaskTopo_t* *taskTopo*, *cnrtUserObject_t* *object*, unsigned int *count*, unsigned int *flags*)

Acquires a reference for a user object from a Task Topo.

Creates or moves user object references that will be owned by a Task Topo.

Parameters

- `[in] taskTopo`: The Task Topo to associate the reference with.
- `[in] object`: The user object to acquire a reference for.
- `[in] count`: The number of references to add to the Task Topo, which is typically 1. The value must be nonzero and not larger than `INT_MAX`.
- `[in] flags`: The optional flag *cnrtTaskTopoUserObjectMove* transfers references from the caller, rather than creating new references, which are created by passing 0.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.17 cnrtTaskTopoReleaseUserObject

cnrtRet_t `cnrtTaskTopoReleaseUserObject(`*cnrtTaskTopo_t* *taskTopo*, *cnrtUserObject_t* *object*, unsigned int *count*)

Releases a reference for a user object from a Task Topo.

Releases user object references owned by a Task Topo.

Parameters

- [in] taskTopo: The Task Topo that will release the reference.
- [in] object: The user object to release a reference for.
- [in] count: The number of references to release, which is typically 1. The value must be nonzero and not larger than INT_MAX.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.18 cnrtTaskTopoAddEmptyNode

```
cnrtRet_t cnrtTaskTopoAddEmptyNode(cnrtTaskTopoNode_t *pNode,
                                   cnrtTaskTopo_t taskTopo,
                                   const cnrtTaskTopoNode_t *dependencies,
                                   size_t numDependencies)
```

Creates an empty node and adds it to a Task Topo.

Creates a new node which performs no operation, and adds it to taskTopo with numDependencies dependencies specified via dependencies. A handle to the new node will be returned in pNode.

An empty node performs no operation during the execution, but it can be transitive ordering. For example, for a phased execution Task Topo with 2 groups of nodes, one group has m nodes, and the other has n nodes, m+n dependency edges are needed with an empty node, while m*n dependency edges are needed without an empty node.

Parameters

- [out] pNode: The value of the granularity returned.
- [in] taskTopo: The properties determine the granularity.
- [in] dependencies: The option to determine the granularity returned. It may not have any duplicate entries.
- [in] numDependencies: The number of dependencies, which can be 0, in which case the node will be placed at the root of the Task Topo.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.19 cnrtTaskTopoAddHostNode

```
cnrtRet_t cnrtTaskTopoAddHostNode(cnrtTaskTopoNode_t *pNode,
                                  cnrtTaskTopo_t taskTopo,
                                  const cnrtTaskTopoNode_t *dependencies,
                                  size_t numDependencies,
                                  const cnrtHostNodeParams_t *nodeParams)
```

Creates a host execution node and adds it to a Task Topo.

Creates a new host API node and adds it to `taskTopo` with `numDependencies` dependencies specified via `dependencies` and arguments specified in `nodeParams`.

Parameters

- `[out] pNode`: The newly created node. When the Task Topo is invoked, the node will invoke the specified CPU function.
- `[in] taskTopo`: The Task Topo to add the node to.
- `[in] dependencies`: The dependencies of the node, which may not have any duplicate entries.
- `[in] numDependencies`: The number of dependencies, which can be 0, in which case the node will be placed at the root of the Task Topo.
- `[in] nodeParams`: Parameters for the host node.

Return *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.20 cnrtTaskTopoHostNodeGetParams

```
cnrtRet_t cnrtTaskTopoHostNodeGetParams(cnrtTaskTopoNode_t node,
                                        cnrtHostNodeParams_t *nodeParams)
```

Returns the parameters of a host node.

Parameters

- `[in] node`: The node to get the parameters for.
- `[out] nodeParams`: Pointer to return the parameters.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.21 cnrtTaskTopoHostNodeSetParams

```
cnrtRet_t cnrtTaskTopoHostNodeSetParams(cnrtTaskTopoNode_t node,
                                        const cnrtHostNodeParams_t *nodeParams)
```

Sets the parameters of a host node.

Sets the parameters of host node `node` to `nodeParams`.

Parameters

- `[in] node`: The node to set the parameters for.
- `[in] nodeParams`: The parameters to copy.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.22 cnrtTaskTopoAddKernelNode

```
cnrtRet_t cnrtTaskTopoAddKernelNode(cnrtTaskTopoNode_t *pNode,
                                    cnrtTaskTopo_t taskTopo,
                                    const cnrtTaskTopoNode_t *dependencies,
                                    size_t numDependencies,
                                    const cnrtKernelNodeParams_t *nodeParams)
```

Creates a kernel execution node and adds it to a Task Topo.

Creates a new kernel execution node and adds it to `taskTopo` with `numDependencies` dependencies specified via `dependencies` and arguments specified in `nodeParams`.

Parameters

- `[out] pNode`: The newly created node.
- `[in] taskTopo`: The Task Topo to add the node to.

- `[in] dependencies`: The dependencies of the node, which may not have any duplicate entries.
- `[in] numDependencies`: The number of dependencies, which can be 0, in which case the node will be placed at the root of the Task Topo.
- `[in] nodeParams`: Parameters for kernel node.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.23 cnrtTaskTopoKernelNodeGetParams

cnrtRet_t `cnrtTaskTopoKernelNodeGetParams`(*cnrtTaskTopoNode_t node*, *cnrtKernelNodeParams_t *nodeParams*)

Returns the parameters of a kernel node.

Returns the parameters of kernel node `node` in `nodeParams`. The `extra` array returned in `nodeParams`, as well as the argument values it points to, are owned by the node.

This memory remains valid until the node is destroyed or its parameters are modified, and should not be modified directly. Use *cnrtTaskTopoKernelNodeSetParams* to update the parameters of this node.

Parameters

- `[in] node`: Node to get the parameters for.
- `[out] nodeParams`: Pointer to return the parameters.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.24 cnrtTaskTopoKernelNodeSetParams

```
cnrtRet_t cnrtTaskTopoKernelNodeSetParams(cnrtTaskTopoNode_t node,
                                          const cnrtKernelNodeParams_t *nodeParams)
```

Sets the parameters of a kernel node.

Sets the parameters of kernel node `node` to `nodeParams`.

Parameters

- `[in] node`: Node to set the parameters for.
- `[in] nodeParams`: Parameters to copy.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.25 cnrtTaskTopoAddMemcpyNode

```
cnrtRet_t cnrtTaskTopoAddMemcpyNode(cnrtTaskTopoNode_t *pNode,
                                    cnrtTaskTopo_t taskTopo,
                                    const cnrtTaskTopoNode_t *dependencies,
                                    size_t numDependencies,
                                    const cnrtMemcpy3dParam_t *copyParams)
```

Creates a memcpy node and adds it to a Task Topo.

Creates a new memcpy node and adds it to `taskTopo` with `numDependencies` dependencies specified via `dependencies`. `numDependencies` can be 0, in which case the node will be placed at the root of the Task Topo. `dependencies` may not have any duplicate entries. A handle to the new node will be returned in `pNode`.

When the Task Topo is invoked, the node will perform the memcpy described by `copyParams`.

See *cnrtMemcpy3D* for the description of the struct and its restrictions.

Currently, memcpy node only supports 1D memcpy, and does not support host to host memory copying. Here is the restriction of `copyParams` on setting 1D memcpy node:

```
#define N copy_size
```

```
cnrtMemcpy3dParam_t memcpy_param = {0};

memcpy_param.dstPtr.pitch = N;
memcpy_param.dstPtr.ysize = 0x1;

memcpy_param.extent.depth = 0x1;
memcpy_param.extent.height = 0x1;
memcpy_param.extent.width = N;

memcpy_param.srcPtr.pitch = N;
memcpy_param.srcPtr.ysize = 0x1;

memcpy_param.src = src_addr;
memcpy_param.dst = dst_addr;
```

Ignores other parameters when setting 1D memcpy node.

Parameters

- [out] pNode: The newly created node.
- [in] taskTopo: The Task Topo to add the node to.
- [in] dependencies: The dependencies of the node.
- [in] numDependencies: The number of dependencies.
- [in] copyParams: The parameters for the memory copy.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.26 cnrtTaskTopoMemcpyNodeGetParams

cnrtRet_t cnrtTaskTopoMemcpyNodeGetParams(*cnrtTaskTopoNode_t node*, *cnrtMemcpy3dParam_t* **nodeParams*)

Returns the parameters of a memcpy node.

Returns the parameters of memcpy node node in nodeParams.

Parameters

- [in] node: The node to get the parameters for.
- [out] nodeParams: Pointer to return the parameters.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.27 cnrtTaskTopoMemcpyNodeSetParams

```
cnrtRet_t cnrtTaskTopoMemcpyNodeSetParams(cnrtTaskTopoNode_t node,
                                          const cnrtMemcpy3dParam_t *nodeParams)
```

Sets the parameters of a memcpy node.

Sets the parameters of memcpy node `node` to `nodeParams`. The restrictions of `nodeParams` are the same as *cnrtTaskTopoAddMemcpyNode()*.

Parameters

- `[in] node`: The node to set the parameters for.
- `[in] nodeParams`: The parameters to copy.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.28 cnrtTaskTopoAddMemsetNode

```
cnrtRet_t cnrtTaskTopoAddMemsetNode(cnrtTaskTopoNode_t *pNode,
                                    cnrtTaskTopo_t taskTopo,
                                    const cnrtTaskTopoNode_t *dependencies,
                                    size_t numDependencies,
                                    const cnrtMemsetParams_t *copyParams)
```

Creates a memset node and adds it to a Task Topo.

Creates a new memset node and adds it to `pNode` with `numDependencies` dependencies specified via `dependencies`. `numDependencies` can be 0, in which case the node will be placed at the root of the Task Topo. `dependencies` may not have any duplicate entries. A handle to the new node will be returned in `pNode`.

The element size must be 1, 2, or 4 bytes. When the Task Topo is invoked, the node will perform the memset described by `copyParams`.

Currently, memset node only supports 1D memset. Set `height` 1 to represent 1D memset.

Parameters

- `[out] pNode`: The newly created node.
- `[in] taskTopo`: The Task Topo to add the node to.
- `[in] dependencies`: The dependencies of the node.
- `[in] numDependencies`: The number of dependencies.
- `[in] copyParams`: The parameters for the memory set.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.29 cnrtTaskTopoMemsetNodeGetParams

```
cnrtRet_t cnrtTaskTopoMemsetNodeGetParams(cnrtTaskTopoNode_t node,
                                          cnrtMemsetParams_t *nodeParams)
```

Returns the parameters of a memset node.

Returns the parameters of memset node `node` in `nodeParams`.

Parameters

- `[in] node`: The node to get the parameters for.
- `[out] nodeParams`: Pointer to return the parameters.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.30 cnrtTaskTopoMemsetNodeSetParams

```
cnrtRet_t cnrtTaskTopoMemsetNodeSetParams(cnrtTaskTopoNode_t node,
                                          const cnrtMemsetParams_t *nodeParams)
```

Sets the parameters of a memset node.

Sets the parameters of memset node `node` to `nodeParams`.

Parameters

- `[in] node`: The node to set the parameters for.
- `[in] nodeParams`: The parameters to copy.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.31 cnrtTaskTopoAddChildTopoNode

```
cnrtRet_t cnrtTaskTopoAddChildTopoNode(cnrtTaskTopoNode_t *pNode,
                                       cnrtTaskTopo_t taskTopo,
                                       const cnrtTaskTopoNode_t *dependencies,
                                       size_t numDependencies,
                                       cnrtTaskTopo_t hChildTopo)
```

Creates a child Task Topo node and adds it to a Task Topo.

Creates a new node which executes an embedded Task Topo, and adds it to `taskTopo` with `numDependencies` dependencies specified via `dependencies`. `numDependencies` can be 0, in which case the node will be placed at the root of the Task Topo. `dependencies` may not have any duplicate entries. A handle to the new node will be returned in `pNode`. The node executes an embedded child Task Topo. The child Task Topo is cloned in this call.

Parameters

- `[out] pNode`: The newly created node.
- `[in] taskTopo`: The Task Topo to add the node to.
- `[in] dependencies`: The dependencies of the node.
- `[in] numDependencies`: The number of dependencies.
- `[in] hChildTopo`: The Task Topo to clone into this node.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.32 cnrtTaskTopoChildTopoNodeGetTopo

```
cnrtRet_t cnrtTaskTopoChildTopoNodeGetTopo(cnrtTaskTopoNode_t node,
                                           cnrtTaskTopo_t *pTaskTopo)
```

Gets a handle to the embedded Task Topo of a child Task Topo node.

This call does not clone the Task Topo. Changes to the Task Topo will be reflected in the node, and the node retains ownership of the Task Topo.

Parameters

- `[in] node`: Node to get the embedded Task Topo for.
- `[out] pTaskTopo`: Location to store a handle to the Task Topo.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.33 cnrtTaskTopoInstantiate

```
cnrtRet_t cnrtTaskTopoInstantiate(cnrtTaskTopoEntity_t *entity,
                                  cnrtTaskTopo_t taskTopo,
                                  cnrtTaskTopoNode_t *pErrorNode,
                                  char *logBuffer,
                                  size_t bufferSize)
```

Creates an executable Task Topo from a Task Topo.

Instantiates `taskTopo` as an executable Task Topo. The Task Topo is validated for any structural constraints or intra-node constraints which were not previously validated. If instantiation is successful, a handle to the instantiated Task Topo is returned in `entity`.

If there are any errors, diagnostic information may be returned in `errorNode` and `logBuffer`. This is the primary way to inspect instantiation errors. The output will be null terminated unless the diagnostics overflow the buffer. In this case, they will be truncated, and the last byte can be inspected to determine if truncation occurs.

Parameters

- `[out] entity`: Returns instantiated Task Topo.
- `[in] taskTopo`: Task Topo to instantiate.
- `[out] pErrorNode`: In case of an instantiation error, this may be modified to indicate a node contributing to the error.
- `[out] logBuffer`: A character buffer to store diagnostic messages.
- `[in] bufferSize`: Size of the log buffer in bytes.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.34 cnrtTaskTopoEntityDestroy

cnrtRet_t `cnrtTaskTopoEntityDestroy(`*cnrtTaskTopoEntity_t entity*`)`

Destroys an executable Task Topo.

Destroys the executable Task Topo specified by `entity`, as well as all of its executable nodes. If the executable Task Topo is being executed, it will not be terminated by this API, while asynchronously released on completion of the Task Topo execution.

Parameters

- `[in] entity`: The executable Task Topo to destroy.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.35 cnrtTaskTopoEntityInvoke

```
cnrtRet_t cnrtTaskTopoEntityInvoke(cnrtTaskTopoEntity_t entity,
                                   cnrtQueue_t queue)
```

Invokes an executable Task Topo in a queue.

Executes `entity` in `queue`. Only one instance of `entity` may be executed at a time. For each invoke, the entity will be ordered after both the previously invoked entity `entity` and the task previously invoked to a queue `queue`. To execute a Task Topo concurrently, it must be instantiated multiple times into multiple executable Task Topo.

Parameters

- `[in] entity`: Executable Task Topo to invoke.
- `[in] queue`: Queue in which to invoke the Task Topo.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*

Note

- None.

Example

- None.

4.9.36 cnrtTaskTopoNodeGetType

```
cnrtRet_t cnrtTaskTopoNodeGetType(cnrtTaskTopoNode_t node,
                                  cnrtTaskTopoNodeType_t *pType)
```

Returns the node type.

Parameters

- `[in] node`: The node to query.
- `[out] pType`: Pointer to return the node type.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.37 cnrtTaskTopoDebugDotPrint

```
cnrtRet_t cnrtTaskTopoDebugDotPrint(cnrtTaskTopo_t taskTopo,
                                    const char *path,
                                    unsigned int flags)
```

Writes a DOT file describing Task Topo struct.

Parameters

- `[in] taskTopo`: The Task Topo to create DOT file from.
- `[in] path`: The path to write the DOT file to.
- `[in] flags`: Flags from *cnrtTaskTopoDebugDotFlags_t* for specifying the additional node information to write.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorSysNoMem*, *cnrtErrorSys*

4.9.38 cnrtTaskTopoKernelNodeGetAttribute

```
cnrtRet_t cnrtTaskTopoKernelNodeGetAttribute(cnrtTaskTopoNode_t node,
                                             cnrtKernelNodeAttr_t attrId,
                                             cnrtKernelNodeAttrValue_t *valueOut)
```

Queries the Task Topo kernel node attribute.

Queries the attribute corresponding to `attrId` from `node`, and stores it in corresponding member of `valueOut`.

Parameters

- `[in] node`: The node to query.
- `[in] attrId`: The attribute ID.
- `[out] valueOut`: The room to store the attribute value.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- The type of `node` must be *cnrtTaskTopoNodeTypeKernel*.

Example

- None.

4.9.39 cnrtTaskTopoKernelNodeSetAttribute

```
cnrtRet_t cnrtTaskTopoKernelNodeSetAttribute(cnrtTaskTopoNode_t node,
                                             cnrtKernelNodeAttr_t attrId,
                                             const cnrtKernelNodeAttrValue_t *value)
```

Sets the Task Topo kernel node attribute.

Sets the attribute corresponding to `attrId` for `node` from corresponding attribute of `value`.

Parameters

- `[in] node`: The node to set attribute for.
- `[in] attrId`: The attribute ID.
- `[in] value`: The attribute value to set.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- The type of `node` must be *cnrtTaskTopoNodeTypeKernel*.

Example

- None.

4.9.40 cnrtTaskTopoKernelNodeCopyAttributes

```
cnrtRet_t cnrtTaskTopoKernelNodeCopyAttributes(cnrtTaskTopoNode_t dst,
                                               cnrtTaskTopoNode_t src)
```

Copies Task Topo kernel node attributes from source node to the destination node.

Parameters

- `[in] dst`: The destination node.
- `[in] src`: The source node.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- Both nodes must belong to the same Task Topo.
- The type of both nodes must be *cnrtTaskTopoNodeTypeKernel*.

Example

- None.

4.9.41 cnrtTaskTopoEntityHostNodeSetParams

```
cnrtRet_t cnrtTaskTopoEntityHostNodeSetParams(cnrtTaskTopoEntity_t entity,
                                              cnrtTaskTopoNode_t node,
                                              const cnrtHostNodeParams_t *nodeParams)
```

Sets the parameters for a host node in the given Task Topo `entity`.

The host node is identified by the corresponding `node` in the non-executable Task Topo, from which the executable Task Topo is instantiated. Changes to to-and-from hNode edges are ignored. The changes only affect future launches of `entity`. Already enqueued or running launches of `entity` are not affected by this call. `node` cannot be modified by this call either.

Parameters

- `[in] entity`: The executable Task Topo in which to set the specified node.
- `[in] node`: Host node of the Task Topo which is used to instantiate entity.
- `[in] nodeParams`: The updated parameters to set.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.42 cnrtTaskTopoEntityKernelNodeSetParams

```
cnrtRet_t cnrtTaskTopoEntityKernelNodeSetParams(cnrtTaskTopoEntity_t entity,
                                                cnrtTaskTopoNode_t node,
                                                const cnrtKernelNodeParams_t *nodeParams)
```

Sets the parameters for a kernel node in the Task Topo `entity`.

The kernel node is identified by the corresponding `node` in the non-executable Task Topo, from which the executable Task Topo entity is instantiated. Changes to to-and-from hNode edges are ignored.

`node` must not have been removed from the original Task Topo entity. The `func` field of `nodeParams` cannot be modified and must match the original value. All other values can be modified.

The changes only affect future launches of `entity`. Already enqueued or running launches of `entity` are not affected by this API. `node` cannot be modified by this API either.

Parameters

- `[in] entity`: The executable Task Topo entity in which to set the specified node.
- `[in] node`: Kernel node of the Task Topo from which Task Topo entity is instantiated.
- `[in] nodeParams`: The updated parameters to set.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.43 cnrtTaskTopoEntityMemcpyNodeSetParams

cnrtRet_t `cnrtTaskTopoEntityMemcpyNodeSetParams`(*cnrtTaskTopoEntity_t entity*, *cnrtTaskTopoNode_t node*, `const` *cnrtMemcpy3dParam_t *nodeParams*)

Sets the parameters for a memcpy node in the given Task Topo entity `entity`.

The memcpy node is identified by the corresponding `node` in the non-executable Task Topo, from which the executable Task Topo entity is instantiated. Changes to to-and-from hNode edges are ignored.

If origin memcpy node is DtoH or HtoD, the device memory must be allocated from the same Context as the original memory. If origin memcpy node is DtoD, the source memory must be allocated from the same Context as the original source memory. Both the instantiation-time memory operands and the memory operands in `nodeParams` must be 1D. Zero-length operations are not supported. The restrictions of `nodeParams` are the same as *cnrtTaskTopoAddMemcpyNode()*.

The changes only affect future launches of `entity`. Already enqueued or running launches of `entity` are not affected by this call. `node` cannot be modified by this call.

Returns *cnrtErrorArgsInvalid* if the memory operands' mappings change; or either the original or new memory operands are multidimensional.

Parameters

- `[in] entity`: The executable Task Topo entity in which to set the specified node.
- `[in] node`: Memcpy node of the Task Topo which is used to instantiate Task Topo entity.
- `[in] nodeParams`: The updated parameters to set.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.44 cnrtTaskTopoEntityMemsetNodeSetParams

```
cnrtRet_t cnrtTaskTopoEntityMemsetNodeSetParams(cnrtTaskTopoEntity_t entity,
                                                cnrtTaskTopoNode_t node,
                                                const cnrtMemsetParams_t
                                                *nodeParams)
```

Sets the parameters for a memset node in the given Task Topo `entity`.

The memset node is identified by the corresponding `node` in the non-executable Task Topo, from which the executable Task Topo entity is instantiated. Changes to to-and-from hNode edges are ignored.

The destination memory in `nodeParams` must be allocated from the same Context as the original destination memory. Both the instantiation-time memory operand and the memory operand in `nodeParams` must be 1D. Zero-length operations are not supported.

The changes only affect future launches of `entity`. Already enqueued or running launches of `entity` are not affected by this call. `node` cannot be modified by this call either.

Returns *cnrtErrorArgsInvalid* if the memory operand's mappings change; or either the original or new memory operand is multi-dimensional.

Parameters

- `[in] entity`: The executable Task Topo entity in which to set the specified node.
- `[in] node`: Memset node of the Task Topo which is used to instantiate Task Topo entity.
- `[in] nodeParams`: The updated parameters to set.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.45 cnrtTaskTopoEntityChildTopoNodeSetParams

```
cnrtRet_t cnrtTaskTopoEntityChildTopoNodeSetParams(cnrtTaskTopoEntity_t entity,
                                                   cnrtTaskTopoNode_t node,
                                                   cnrtTaskTopo_t childTopo)
```

Updates node parameters in the child Task Topo node in the given Task Topo `entity`.

The child node is identified by the corresponding `node` in the non-executable Task Topo, from which the executable Task Topo entity is instantiated. Changes to to-and-from hNode edges are ignored.

The topology of `childTopo`, as well as the node insertion order, must match that of the Task Topo contained in `node`. See *cnrtTaskTopoEntityUpdate()* for a list of restrictions on what can be updated in Task Topo entity. The update is recursive, so child Topo nodes contained within the top level child Task Topo will also be updated.

The changes only affect future launches of `entity`. Already enqueued or running launches of `entity` are not affected by this call. `node` cannot be modified by this call either.

Parameters

- `[in] entity`: The executable Task Topo entity in which to set the specified node.
- `[in] node`: Child node of the Task Topo which is used to instantiate Task Topo entity.
- `[in] childTopo`: The child Task Topo supplying the updated parameters.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*

Note

- None.

Example

- None.

4.9.46 cnrtTaskTopoEntityUpdate

```
cnrtRet_t cnrtTaskTopoEntityUpdate(cnrtTaskTopoEntity_t entity,
                                   cnrtTaskTopo_t topo,
                                   cnrtTaskTopoNode_t *pErrorNode_out,
                                   cnrtTaskTopoEntityUpdateResult_t
                                   *updateResult_out)
```

Checks whether an executable Task Topo entity can be updated with a Task Topo and performs the update accordingly.

Updates the node parameters in the instantiated Task Topo entity specified by `entity` with the node parameters in a topologically identical Task Topo specified by `topo`.

Limitations:

- Kernel node restrictions:
  - The owning Context of the node cannot be changed.
- Memset node restrictions:
  - The device(s) to which the operand(s) is allocated/mapped cannot be changed.
  - The memory must be allocated from the same Context as the original memory.
  - Only 1D memset is supported now.
  - Zero-length operations are not supported.
- Memcpy node restrictions:
  - If memcpy is DtoH or HtoD, the device memory must be allocated from the same Context as the original memory.
  - If memcpy is DtoD, the source memory must be allocated from the same Context as the original source memory.
  - Only 1D memcpy is supported now.
  - Zero-length operations are not supported.

cnrtTaskTopoEntityUpdate() sets `updateResult_out` to *cnrtTaskTopoEntityUpdateErrorTopologyChanged* under the following conditions:

- The count of nodes directly in `entity` and `node` differ, in which case `updateResult_out` is NULL.
- A node is deleted in `node` but not its pair from `entity`, in which case `updateResult_out` is NULL.
- A node is deleted in `entity` but not its pair from `node`, in which case `updateResult_out` is the pairless node from `node`.
- The dependent nodes of a pair differ, in which case `updateResult_out` is the node from `topo`.

cnrtTaskTopoEntityUpdate() sets `updateResult_out` to:

- *cnrtTaskTopoEntityUpdateError* if an invalid value is passed.
- *cnrtTaskTopoEntityUpdateErrorTopologyChanged* if the Task Topo topology is changed.
- *cnrtTaskTopoEntityUpdateErrorNodeTypeChanged* if the type of a node is changed, in which case `updateResult_out` is set to the node from `topo`.
- *cnrtTaskTopoEntityUpdateErrorUnsupportedFunctionChange* if the function changes in an unsupported way (see note above), in which case `pErrorNode_out` is set to the node from `topo`.
- *cnrtTaskTopoEntityUpdateErrorParametersChanged* if any parameter of a node is changed in a way that is not supported, in which case `pErrorNode_out` is set to the node from `topo`.
- *cnrtTaskTopoEntityUpdateErrorAttributesChanged* if any attribute of a node is changed in a way that is not supported, in which case `pErrorNode_out` is set to the node from `topo`.

- *cnrtTaskTopoEntityUpdateErrorNotSupported* if something about a node is unsupported, in which case pErrorNode_out is set to the node from topo.

If updateResult_out isn't set in one of the situations described above, the update check passes and *cnrtTaskTopoEntityUpdate()* updates entity to match the contents of topo. If an error occurs during the update, updateResult_out will be set to *cnrtTaskTopoEntityUpdateError*; otherwise, updateResult_out is set to *cnrtTaskTopoEntityUpdateSuccess*.

cnrtTaskTopoEntityUpdate() returns *cnrtSuccess* when the update is performed successfully. It returns *cnrtErrorTaskTopoEntityUpdateFailure* if the Task Topo entity update is not performed because it includes changes which violate constraints specific to instantiated Task Topo entity update.

Parameters

- [in] entity: The instantiated Task Topo entity to be updated.
- [in] topo: The Task Topo containing the updated parameters.
- [out] pErrorNode_out: The node which causes the permissibility check to forbid the update, if any.
- [out] updateResult_out: Whether the Task Topo update is permitted, and what the reason is if it is forbidden.

Return

- *cnrtSuccess*, *cnrtErrorArgsInvalid*, *cnrtErrorTaskTopoEntityUpdateFailure*

Note

- The API may add further restrictions in future releases. The return code should always be checked.

Example

- None.