User Target Kernel Extension

The Khronos® OpenVX Working Group, Editors: Raphael Cano, Jesse Villarreal, Lucas Weaver

Version 1.0, Tue Sep 30 2025 17:31:08: 6419f26-dirty

Chapter 1

User Target Kernel Extension

1.1 Introduction

1.1.1 Purpose

This document details an extension to any OpenVX version from 1.1 to 1.3, and references some APIs and symbols that may be found in those APIs: `https://www.khronos.org/registry/OpenVX/`.

This extension is intended to define support for the user kernels which can be executed on remote targets in the system, not just the host core.

1.1.2 Acknowledgements

This specification would not be possible without the contributions from this partial list of the following individuals from the Khronos Working Group and the companies that they represented at the time:

- Kiriti Nagesh Gowda - AMD
- Raphael Cano - Robert Bosch GmbH
- Jesse Villarreal - TI
- Isaac Wong - Ambarella International LP
- Viktor Gyenes - AI Motive

1.1.3 Background

The main OpenVX specification includes APIs to support `User Kernels`, which enables users to plugin their own kernels. However, this support is limited to user kernels which are **executed on the host core**, including host core calls into an API such as OpenCL/Vulkan/OpenGL to dispatch to a hardware accelerator such as a GPU.

Some systems include a variety of heterogeneous CPUs and hardware accelerators which can be leveraged by nodes in OpenVX graphs. In such systems, the host can offload processing to dedicated hardware targets in the system, freeing up cycles on the host processor for other tasks. This extension provides support for users to add kernels on other targets in such systems.

The diagram below shows an example system which has several targets where user kernels can run. The left side of this diagram can make use of the existing user kernel functionality from the main specification, where the host CPU can either run CPU-based kernels, or potentially run an OpenCL kernel which can offload to a GPU, for example. The right side of this diagram shows how OpenVX nodes can be executed on remote CPUs. These remote CPUs can execute kernels which can run algorithms or host drivers to hardware accelerators.

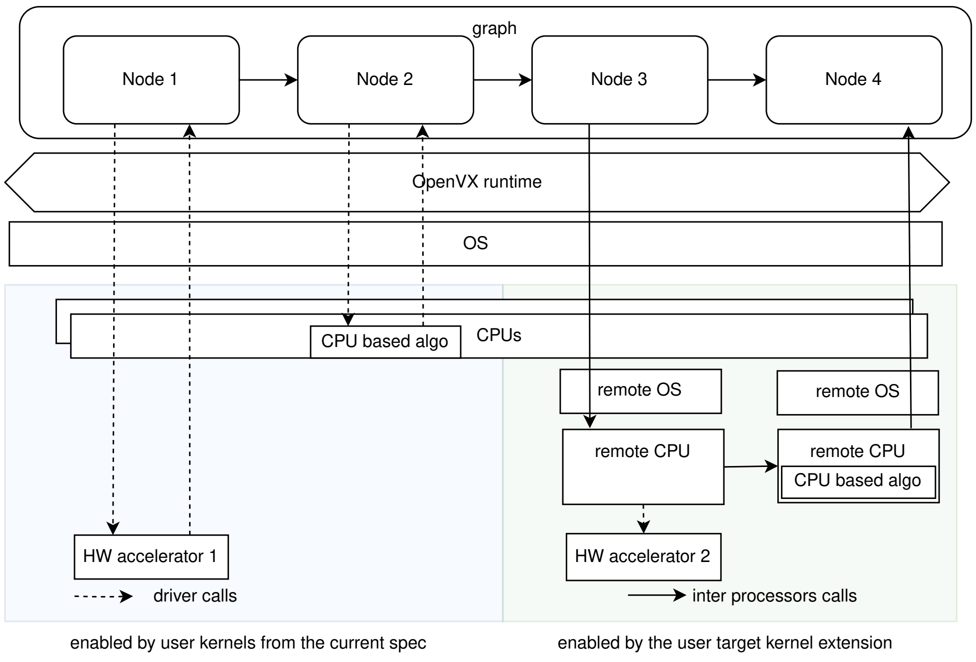

Figure 1.1 Heterogeneous SoC Model

For this purpose, a new set of "Target Kernel" APIs has been designed.

1.2 Design Overview

1.2.1 User Target Kernel Callbacks

This extension builds on the existing pattern from the OpenVX specification. In the main specification, the `vxAddUserKernel` API is defined, which allows the user to register the kernel (by name and enum) to the framework, along with four kernel callback functions which are compiled and linked to run on the host processor:

- **init**: Kernel initialization function
- **validate**: Validation function, which validates parameters to the kernel
- **func_ptr**: Main processing function which is called each time the graph is executed.
- **deinit**: Kernel deinitialization function

This extension also leverages this same function to register the kernel and host-side callbacks on the host CPU, however, if the user intends this kernel to execute on a remote target instead of the host CPU, the user should pass a **NULL** pointer to the **func_ptr** callback pointer. This signals the framework that this kernel is expected to also be registered on a remote CPU in the system using the vxAddTargetKernel or vxAddTargetKernelByName APIs, which registers up to four additional target-side callbacks:

- **create_func**: Called during graph verification, to perform any local memory setup or one-time configuration.
- **process_func**: Main processing function which is called each time the graph is executed.
- **control_func**: Can optionally be called asynchronously via `vxNodeSendCommand` from the application.
- **delete_func**: Called during graph release, to release local memory or tear-down any local setup.

The following diagram depicts these two sets of callbacks: one for the host, and one for the target. The reason for the separation of these callbacks and associated registration functions is due to the fact that the target callbacks typically are linked into the executable binaries of one or more remote core firmwares separate from the host callbacks.

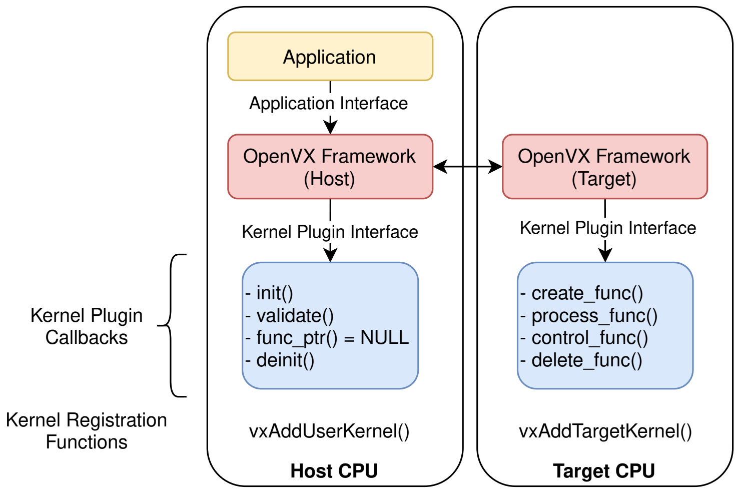

Figure 1.2 User Kernel Callbacks

During graph verification, before the creation of individual nodes on the remote core(s) (see the relevant callbacks), the host may validate the kernel parameters used to create each node. For this validation, the host may need to access the complete set of OpenVX objects and this can be done within the original user node validate callback which resides on the host.

The following call sequence shows the relative interaction between the host application and the target kernel callbacks:

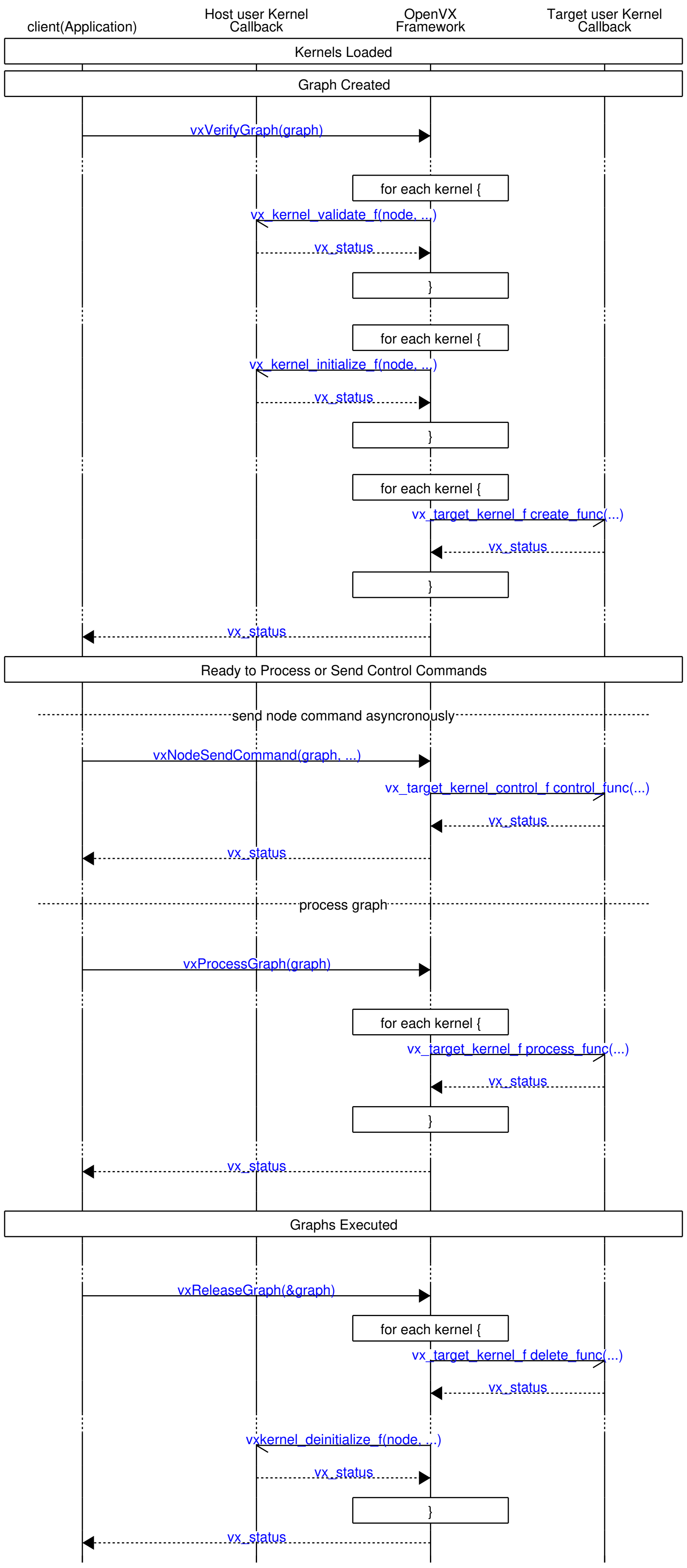
client(Application)
Host user Kernel Callback
OpenVX Framework
Target user Kernel Callback
Kernels Loaded
Graph Created
vxVerifyGraph(graph)
for each kernel {
vx_kernel_validate_f(node, ...)
vx_status
}
for each kernel {
vx_kernel_initialize_f(node, ...)
vx_status
}
for each kernel {
vx_target_kernel_f create_func(...)
vx_status
}
vx_status
Ready to Process or Send Control Commands
send node command asyncronously
vxNodeSendCommand(graph, ...)
vx_target_kernel_control_f control_func(...)
vx_status
vx_status
process graph
vxProcessGraph(graph)
for each kernel {
vx_target_kernel_f process_func(...)
vx_status
}
vx_status
Graphs Executed
vxReleaseGraph(&graph)
for each kernel {
vx_target_kernel_f delete_func(...)
vx_status
vxkernel_deinitialize_f(node, ...)
vx_status
}
vx_status

Figure 1.3 Sequence of Host/Target Kernel Interaction

1.2.2 User Target Kernel Registration

As mentioned earlier, the `vxAddUserKernel` function requires both a kernel name and a kernel enumeration on the host. The enumeration may be statically defined, as in the case for the standard OpenVX vision kernels, or it may be dynamically allocated using `vxAllocateUserKernelId`. These two ways of obtaining kernel enumerations are the reason why this extension provides two separate options for registering user target kernels as indicated below:

- vxAddTargetKernel : Can be used when the kernel enumeration is known on the target CPU at the time of target kernel registration (for example, if it is statically assigned at build time)
- vxAddTargetKernelByName : Should be used when the kernel enumeration is **NOT** known on the target CPU at the time of target kernel registration (for example, if it is dynamically assigned at run time on the host via `vxAllocateUserKernelId`).

Kernel names are usually statically defined, so technically vxAddTargetKernelByName can also be used instead of vxAddTargetKernel even if the kernel enumeration is known on the target CPU. However, both functions are available in case a framework or kernel implementation is more optimal using vxAddTargetKernel when it can.

1.2.3 Target-side Opaque Objects

As illustrated in the Heterogeneous SoC Model from the "Background" section, the host and target model requires sharing only a minimal set of data to allow the remote kernel to access essential information, such as:

- Inputs
- Outputs
- Kernel parameters
- Other relevant metadata

While it is technically possible to exchange complete set of OpenVX objects between the host and targets, doing so may significantly increase memory usage and result in unnecessary data transfer. To address this, this extension prioritizes minimizing the memory footprint by exchanging only the essential subset of information:

- vx_target_kernel : Opaque target kernel object (target-side equivalent of vx_kernel object)
  - Used when adding and removing the target kernel
- vx_target_kernel_instance : Opaque target kernel instance passed to the kernel callbacks (target-side equivalent of vx_node object)
  - Used, if needed, to set and get the kernel instance context that can be shared between callbacks (described below)
- vx_object_desc : Opaque target input and output data object information (target-side equivalent to vx_↩ reference object for data objects)
  - Used to access node parameters within the target callbacks
  - Must be shared between the host and (possibly multiple) remote cores in a manner that prevents concurrent read/write access
- The number of parameters.
- And some additional optional data.

These objects are designed to contain just enough data to enable the invocation and execution of the four callback functions described earlier, specifically on a dedicated target. As a trade-off, the target kernel will not have access to the full set of OpenVX APIs. However, this is not required, since kernel implementations do not interact with the graph or other abstract objects.

1.2.3.1 Accessing data from Objects

The actual contents of vx_target_kernel_instance and vx_object_desc, and the mechanism by which the writers of the user target kernels can access required information (such as image width/height, buffer addresses, etc) are not specified in this extension. These details are implementation dependent. The rationale for this are as follows:

- The priority for this extension is a lightweight implementation on targets to optimize for memory and speed. Therefore, reusing the existing data object access functions, or creating target-side equivalents may violate this priority.
- Unlike host-only user kernels, target kernels are more often then not, vendor specific. For example, they include categories of algorithms optimized for remote targets like specific DSPs, or hardware drivers specific to vendors' IP. Therefore, portability of the actual user target kernels is not as high of a priority for this extension as compared to remote core code size.

Therefore, this extension focuses on providing the high level mechanism of registration functions for specific callbacks, saving and accessing context between callbacks, and memory allocations specific for the target.

1.2.4 Target Kernel Instance Context

In many cases, there may be some context information that needs to be shared between the callbacks of a kernel. For example, if a kernel instance needs some scratch memory, the create callback can allocate it, the process callback can utilize it, and the delete callback can free it. Additionally, there may be some parameters that are calculated or setup in the create callback, and are tracked and updated in the process callback. This context is kernel-specific, typically using a custom structure that can contain all the pointers, sizes, and parameters that needs to be maintained and shared between the callbacks. Once a buffer of the size of this context structure is allocated in the create callback, it can be initialized and then finally registered and retrieved using the following calls:

- vxSetTargetKernelInstanceContext typically called from the create callback to save off the pointer to an instance context that can be retrieved from other callbacks
- vxGetTargetKernelInstanceContext called from kernel callbacks to retrieve the kernel instance context for reading or updating

1.2.5 Memory Allocation

OpenVX does not dictate any requirements on memory allocation methods or the layout of opaque memory objects and it does not dictate byte packing or alignment for structures on architectures. This extension introduces a feature that allows users to implement specific functions for dedicated hardware accelerators or CPUs, which may run remotely on separate operating systems and hardware infrastructures. In such cases, it can be beneficial to allocate "local" memory dedicated to these subsystems.

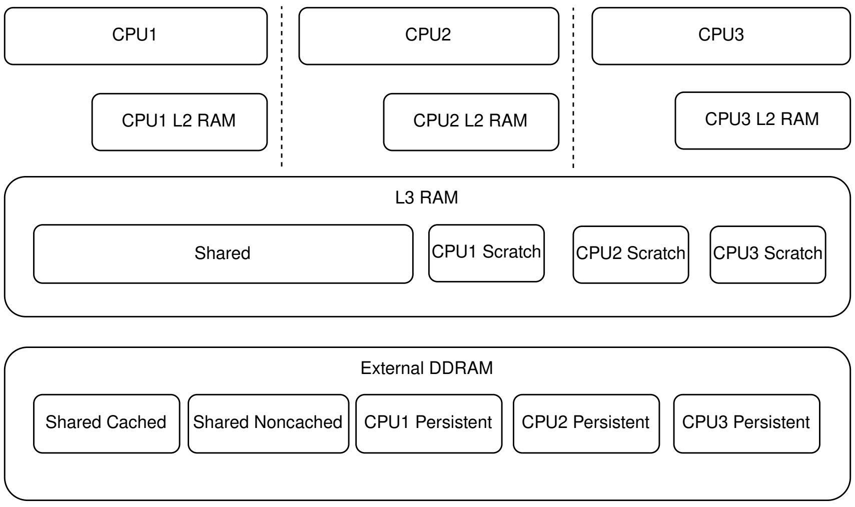

Figure 1.4 Example SoC Memory Hierarchy Diagram

Therefore, it is necessary to add a standard generic OpenVX memory allocator to support new types of memory pools tailored for target-specific allocations. This enables efficient and flexible memory management for remote cores, ensuring that each subsystem can utilize memory resources optimized for its unique requirements. This extension introduces a memory allocator function, vxMemTargetAlloc, which is intended for exclusive use by the user node. It allows the allocation of memory blocks of specific sizes from designated memory pools.

This extension only defines a single generic enumeration to be used with this allocator: VX_MEM_POOL_ANY. Since the specific available memory pools in a system are highly system specific, it is expected that a vendor may define extension memory pool enumerations which can be used by user kernels implemented for their systems.

For example, the following table could be an example of a vendor extension list of memory pools based on the arbitrary example given in the above diagram: Example SoC Memory Hierarchy Diagram

Note

This table is just an example for how a specific system may choose to define different mempools.

Mempool Categories	Mempool Name	Comments
Standard/Portable Memory Pool	VX_MEM_POOL_ANY	Unspecified memory pool
CPU-Dedicated Memory Pools (memory mapped to specific cores they are dedicated to)	XYZ_MEM_POOL_L2	Memory pool associated with the L2 RAM of a specific CPU core
	XYZ_MEM_POOL_L3_SCRATCH	Scratchpad memory in L3 RAM, typically used for temporary or high-speed data storage
	XYZ_MEM_POOL_EXT_↩ PERSISTENT	External persistent memory pool for long-term data storage
Remote/Shared Memory Pools (Memory pools shared across remote cores or systems)	XYZ_MEM_POOL_L3_SHARED	Shared L3 cache memory pool accessible by remote cores/systems
	XYZ_MEM_POOL_EXT_↩ SHARED_CACHED	External shared memory pool with caching enabled
	XYZ_MEM_POOL_P_EXT_↩ SHARED_NONCACHED	External shared memory pool with caching disabled

During release graph, the memory allocated on each remote core must be freed via the vxMemTargetFree within the corresponding node **delete_func** callback.

1.3 Kernel Callback Developer Guidelines

When a framework includes user callbacks, there are usually assumptions that the framework makes about how those callbacks are implemented. OpenVX is no exception. The following sections contain guidelines and assumptions that User Target Kernel callback implementers should follow for proper usage when using both the default behavior of the OpenVX framework as well as some additional considerations when using graph or node level timeouts.

1.3.1 create_func

1.3.1.1 Thread/blocking Implications

The `vxVerifyGraph` function is a blocking function which runs to completion before returning. It calls the **create_func** callback for each node and then doesn't return until all node create function callbacks return. Therefore, the following guidelines shall be followed:

- There shall not be any dependency on another node's create callback since it may not have executed yet in the sequence of calls to each node.
- There shall not be any dependency on some action that the application does after returning from `vx←VerifyGraph`. For example, a blocking call called from within the **create_func** will result in blocking the full `vxVerifyGraph`, potentially causing a deadlock if the **create_func** is waiting for further action from the same thread in the application which called `vxVerifyGraph`, or from another node's create function.

1.3.1.2 Memory Implications

If there is some context which needs to be accessed for the other target-side callbacks for a given kernel, it should be created in the **create_func** since memory allocations are not allowed in any callback except the **create_func** callback.

- The context pointer can be allocated using vxMemTargetAlloc. This allocator allows the user to allocate memory on the remote core where the kernel will run and will be seen by this specific kernel instance only. The following is an example of the allocation of the vxCannyParams data structure context:
```
vxCannyParams prms = vxMemTargetAlloc(sizeof(vxCannyParams), VX_MEM_POOL_ANY);
```
- The allocated context shall be added to the target kernel instance using the vxSetTargetKernelInstanceContext function (so the other callbacks can retrieve it via vxGetTargetKernelInstanceContext. For example for the canny edge parameters:
```
vx_status status = vxSetTargetKernelInstanceContext(kernel, prms, sizeof(vxCannyParams));
```
- If the node instance needs additional memory, then it should allocate it in the **create_func** callback using the vxMemTargetAlloc with the appropriate memory pool needed based on what is made available in the system by the vendor. Then the corresponding pointers and sizes can be added to the context structure to be accessed by the other callbacks.

1.3.2 process_func

1.3.2.1 Thread/blocking Implications

The **process_func** callback is called for each node in order of graph dependency. Therefore, upon returning from a process function, the framework shall assume that the operations are complete and the inputs are no longer being read, and the outputs are no longer being updated. Therefore the process function should ensure completion of the job before returning.

1.3.2.2 Memory Implications

- No memory allocations should be made in the **process_func** callback. Any memory allocations should have been created in the **create_func** (see above).
- If there is some context which needs to be accessed/updated from the **create_func** callback, it can be retrieved from the kernel instance using the vxGetTargetKernelInstanceContext function:
```
status = vxGetTargetKernelInstanceContext(kernel, (void **)&prms, &size);
```

1.3.3 control_func

1.3.3.1 Data Object Verification Implications

Since the objects being used with the control callback are not necessarily node parameters, the parameters are not subject to the validate callback checks being done during the call to `vxVerifyGraph`. Therefore, if an object is to be used within a control callback, it is necessary for the control callback (or some other mechanism within the application, etc) to perform validation of the parameters being used within the callback.

1.3.3.2 Thread/blocking Implications:

The **control_func** callback (if implemented) is triggered from the application by calling `vxNodeSendCommand`. The call to `vxNodeSendCommand` is blocked until the target can complete execution of the corresponding **control_func** callback.

Note

Since the call to `vxNodeSendCommand` is made asynchronous to the **process_func**, there is no guarantee on the order or exact time the command will get executed (i.e. it could get executed a few frames after it was called depending on the implementation).

1.3.3.3 Memory Implications

- No memory allocations should happen in the **control_func** callback. Any memory allocations should have been created in the **create_func** (see above).
- If there is some context which needs to be accessed/updated from the **create_func** callback, it can be retrieved from the kernel instance using the vxGetTargetKernelInstanceContext function:
```
status = vxGetTargetKernelInstanceContext(kernel, (void **)&prms, &size);
```
- The **control_func** callback (if implemented) should only be called after `vxVerifyGraph` and before `vx↩ReleaseGraph`, since it may need to access the kernel instance context, which only exists in the time between these two calls.

1.3.4 delete_func

1.3.4.1 Thread/blocking Implications

The `vxReleaseGraph` function is a blocking function which runs to completion before returning. It calls the **delete_func** callback for each node one by one (sequentially) and then doesn't return until all node delete function callbacks return. Therefore, the following guidelines shall be followed:

- There shall not be any dependency on another node's delete function since it may not have executed yet in the sequence of calls to each node.
- There shall not be any dependency on some action that the application does after returning from `vx↩ReleaseGraph`. For example, a blocking call called from within the **delete_func** will result in blocking the full `vxReleaseGraph`, potentially causing a deadlock if the **delete_func** is waiting for further action from the same thread in the application which called `vxReleaseGraph`, or from another node's delete function.

1.3.4.2 Memory Implications

All memory buffers allocated during the **create_func** should be freed in the **delete_func**:

- If there is some context which was allocated in the **create_func** callback, it can be retrieved from the kernel instance using the vxGetTargetKernelInstanceContext function:
  ```
  status = vxGetTargetKernelInstanceContext(kernel, (void **)&prms, &size);
  ```
- If the kernel instance context included pointers/sizes to additional scratch or persistent memory allocated in the **create_func** callback, it should be freed in the **delete_func** callback using the vxMemTargetFree function with the dedicated mempool.
- If the **create_func** allocated kernel instance context, it should be freed from the kernel instance using the vxMemTargetFree function:
  ```
  vxMemTargetFree(prms, sizeof(vxCannyParams), VX_MEM_POOL_ANY);
  ```

Chapter 2

Requirements

Global VX_MEM_POOL_ANY

[REQ-USERKERNEL-01]: VX_MEM_POOL_ANY

Global vx_object_desc

[REQ-USERKERNEL-02]: vx_object_desc

Global vx_target_kernel

[REQ-USERKERNEL-03]: vx_target_kernel

Global vx_target_kernel_control_f)(vx_target_kernel_instance kernel, vx_uint32 node_cmd_id, vx_object_desc obj_desc[], vx_uint16 num_params, void ∗priv_arg)

[REQ-USERKERNEL-08]: vx_target_kernel_control_f

Global vx_target_kernel_f)(vx_target_kernel_instance kernel, vx_object_desc obj_desc[], vx_uint16 num←_params, void ∗priv_arg)

[REQ-USERKERNEL-07]: vx_target_kernel_f

Global vx_target_kernel_instance

[REQ-USERKERNEL-04]: vx_target_kernel_instance

Global vxAddTargetKernel (vx_enum kernel_id, const vx_char ∗target_name, vx_target_kernel_f process←_func, vx_target_kernel_f create_func, vx_target_kernel_f delete_func, vx_target_kernel_control_f control_func, void ∗priv_arg)

[REQ-USERKERNEL-09]: vxAddTargetKernel

Global vxAddTargetKernelByName (const vx_char ∗kernel_name, const vx_char ∗target_name, vx_target_kernel_f process_func, vx_target_kernel_f create_func, vx_target_kernel_f delete_func, vx_target_kernel_control_f control_func, void ∗priv_arg)

[REQ-USERKERNEL-10]: vxAddTargetKernelByName

Global vxGetTargetKernelInstanceContext (vx_target_kernel_instance target_kernel_instance, void ∗∗kernel_context, vx_uint32 ∗kernel_context_size)

[REQ-USERKERNEL-14]: vxGetTargetKernelInstanceContext

Global vxMemTargetAlloc (vx_uint32 size, vx_enum mem_pool)

[REQ-USERKERNEL-05]: vxMemTargetAlloc

Global vxMemTargetFree (void ∗ptr, vx_uint32 size, vx_enum mem_pool)

[REQ-USERKERNEL-06]: vxMemTargetFree

Global vxRemoveTargetKernel (vx_target_kernel target_kernel)

[REQ-USERKERNEL-11]: vxRemoveTargetKernel

Global vxRemoveTargetKernelByName (const vx_char ∗kernel_name, const vx_char ∗target_name)

[REQ-USERKERNEL-12]: vxRemoveTargetKernelByName

Global vxSetTargetKernelInstanceContext (vx_target_kernel_instance target_kernel_instance, void ∗kernel_context, vx_uint32 kernel_context_size)

[REQ-USERKERNEL-13]: vxSetTargetKernelInstanceContext

Chapter 3

API Documentation

3.1 User Target Kernel Extension

Typedefs

- typedef struct _vx_object_desc ∗ vx_object_desc
- typedef struct _vx_target_kernel ∗ vx_target_kernel
- typedef struct _vx_target_kernel_instance ∗ vx_target_kernel_instance
- typedef vx_status(∗ vx_target_kernel_f) (vx_target_kernel_instance kernel, vx_object_desc obj_desc[ ], vx↩ _uint16 num_params, void ∗priv_arg)
- typedef vx_status(∗ vx_target_kernel_control_f) (vx_target_kernel_instance kernel, vx_uint32 node_cmd_id, vx_object_desc obj_desc[ ], vx_uint16 num_params, void ∗priv_arg)

Enumerations

- enum vx_target_kernel_mem_pool_enum_e
- enum vx_target_kernel_mem_pool_type_e

Functions

- void ∗ vxMemTargetAlloc (vx_uint32 size, vx_enum mem_pool)
- void vxMemTargetFree (void ∗ptr, vx_uint32 size, vx_enum mem_pool)
- vx_target_kernel vxAddTargetKernel (vx_enum kernel_id, const vx_char ∗target_name, vx_target_kernel_f process_func, vx_target_kernel_f create_func, vx_target_kernel_f delete_func, vx_target_kernel_control_f control_func, void ∗priv_arg)
- vx_target_kernel vxAddTargetKernelByName (const vx_char ∗kernel_name, const vx_char ∗target_↩ name, vx_target_kernel_f process_func, vx_target_kernel_f create_func, vx_target_kernel_f delete_func, vx_target_kernel_control_f control_func, void ∗priv_arg)
- vx_status vxRemoveTargetKernel (vx_target_kernel target_kernel)
- vx_status vxRemoveTargetKernelByName (const vx_char ∗kernel_name, const vx_char ∗target_name)
- vx_status vxSetTargetKernelInstanceContext (vx_target_kernel_instance target_kernel_instance, void ∗kernel_context, vx_uint32 kernel_context_size)
- vx_status vxGetTargetKernelInstanceContext (vx_target_kernel_instance target_kernel_instance, void ∗∗kernel_context, vx_uint32 ∗kernel_context_size)

3.1.1 Detailed Description

This section lists the APIs required for User Target Kernels.

3.1.2 Typedef Documentation

3.1.2.1 vx_object_desc

```
typedef struct _vx_object_desc* vx_object_desc
```

A generic opaque reference that encapsulates all data necessary for node execution on a target hardware.

the object must be shared between the host and (possibly multiple) remote cores in a manner that prevents concurrent read/write access

Requirement [REQ-USERKERNEL-02]: vx_object_desc

Definition at line 65 of file vx_khr_target_kernel.h.

3.1.2.2 vx_target_kernel

```
typedef struct _vx_target_kernel* vx_target_kernel
```

Handle to kernel on a target.

Requirement [REQ-USERKERNEL-03]: vx_target_kernel

Definition at line 73 of file vx_khr_target_kernel.h.

3.1.2.3 vx_target_kernel_instance

```
typedef struct _vx_target_kernel_instance* vx_target_kernel_instance
```

Handle to instance of a kernel on a target.

Requirement [REQ-USERKERNEL-04]: vx_target_kernel_instance

Definition at line 80 of file vx_khr_target_kernel.h.

3.1.2.4 vx_target_kernel_f

```
typedef vx_status( * vx_target_kernel_f) (vx_target_kernel_instance kernel, vx_object_desc
obj_desc[ ], vx_uint16 num_params, void *priv_arg)
```

The target kernel callbacks prototype.

For create_func, delete_func, and process_func callbacks 'obj_desc' points to array of data object descriptor parameters

Parameters

in	*kernel*	The kernel for which the callback is called
in	*obj_desc*	Object descriptor passed as input to this callback
in	*num_params*	valid entries in object descriptor (obj_desc) array
in	*priv_arg*	additional private argument passed to the callback

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors; any other value indicates failure.

Requirement [REQ-USERKERNEL-07]: vx_target_kernel_f

Definition at line 134 of file vx_khr_target_kernel.h.

3.1.2.5 vx_target_kernel_control_f

```
typedef vx_status( * vx_target_kernel_control_f) (vx_target_kernel_instance kernel, vx_uint32
node_cmd_id, vx_object_desc obj_desc[ ], vx_uint16 num_params, void *priv_arg)
```

The target kernel callback for control command.

Used for control_func, 'obj_desc' points to array of objects descriptors for control parameter. It could be any vx_↩
(object)

Parameters

in	*kernel*	The kernel for which the callback is called
in	*node_cmd↩ _id*	Command ID to be processed in the given node
in	*obj_desc*	Object descriptor passed as input to this callback
in	*num_params*	valid entries in object descriptor (obj_desc) array
in	*priv_arg*	additional private argument passed to the callback

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors; any other value indicates failure.

Requirement [REQ-USERKERNEL-08]: vx_target_kernel_control_f

Definition at line 158 of file vx_khr_target_kernel.h.

3.1.3 Enumeration Type Documentation

3.1.3.1 vx_target_kernel_mem_pool_enum_e

```
enum vx_target_kernel_mem_pool_enum_e
```

Extra enums.

Enumerator

VX_ENUM_MEM_POOL	Memory pool type enumeration.

Definition at line 36 of file vx_khr_target_kernel.h.

3.1.3.2 vx_target_kernel_mem_pool_type_e

```
enum vx_target_kernel_mem_pool_type_e
```

Type of memory pool.

See `vxMemTargetAlloc` and `vxMemTargetFree`

Enumerator

VX_MEM_POOL_ANY	Allocate memory in any memory pool. **Requirement** [REQ-USERKERNEL-01]: VX_MEM_POOL_ANY

Definition at line 47 of file vx_khr_target_kernel.h.

3.1.4 Function Documentation

3.1.4.1 vxMemTargetAlloc()

```
void * vxMemTargetAlloc (
            vx_uint32 size,
            vx_enum mem_pool)
```

Allocates memory of given size in the specified memory pool on a target.

Parameters

in	*size*	size of the memory to be allocated
in	*mem_pool*	dedicated memory pool to allocate from

See also

vx_target_kernel_mem_pool_type_e

memory allocator function, which is intended for exclusive use by the user node. It allows the allocation of memory blocks of specific sizes from designated memory pools. The memory allocation should happen during the node create phase

Returns

Pointer to the allocated memory

Requirement [REQ-USERKERNEL-05]: vxMemTargetAlloc

3.1.4.2 vxMemTargetFree()

```
void vxMemTargetFree (
            void * ptr,
            vx_uint32 size,
            vx_enum mem_pool)
```

Frees already allocated memory.

Parameters

in	*ptr*	Pointer to the memory
in	*size*	size of the memory to be freed
in	*mem_pool*	Memory pool from which the memory was allocated

See also

vx_target_kernel_mem_pool_type_e

During release graph, the memory allocated on each remote core must be freed by the corresponding node during the node delete phase.

Requirement [REQ-USERKERNEL-06]: vxMemTargetFree

3.1.4.3 vxAddTargetKernel()

```
vx_target_kernel vxAddTargetKernel (
            vx_enum kernel_id,
            const vx_char * target_name,
            vx_target_kernel_f process_func,
            vx_target_kernel_f create_func,
            vx_target_kernel_f delete_func,
            vx_target_kernel_control_f control_func,
            void * priv_arg)
```

Allows users to add native kernels implementation to specific targets.

This is different from vxAddUserKernel() in that this is called on the target CPU and it allows users to implement plugin specific kernels An equivalent vxAddUserKernel is typically called to pair the target kernel with OpenVX user kernel.

Same as vxAddTargetKernelByName except that it take a kernel_id as input instead of a string name

Parameters

in	*kernel_id*	Unique identifier for the kernel, based on the vx_kernel_e enumeration
in	*target_name*	Name of the target
in	*process_func*	Function pointer for the kernel processing function
in	*create_func*	Function pointer for the kernel creation function
in	*delete_func*	Function pointer for the kernel deletion function
in	*control_func*	Function pointer for the kernel control function
in	*priv_arg*	Private argument passed to the kernel

Returns

A target kernel reference.

Requirement [REQ-USERKERNEL-09]: vxAddTargetKernel

3.1.4.4 vxAddTargetKernelByName()

```
vx_target_kernel vxAddTargetKernelByName (
            const vx_char * kernel_name,
            const vx_char * target_name,
            vx_target_kernel_f process_func,
            vx_target_kernel_f create_func,
            vx_target_kernel_f delete_func,
            vx_target_kernel_control_f control_func,
            void * priv_arg)
```

Allows users to add native kernels implementation to specific targets.

This is different from vxAddUserKernel() in that this is called on the target CPU and it allows users to implement plugin specific kernels An equivalent vxAddUserKernel is typically called to pair the target kernel with OpenVX user kernel.

Same as vxAddTargetKernel except that it take a string name as input instead of kernel_id

Important Note: The user must ensure all kernel names are unique on a given core.

Parameters

in	*kernel_name*	Name of the target kernel
in	*target_name*	Name of the target
in	*process_func*	Function pointer for the kernel processing function
in	*create_func*	Function pointer for the kernel creation function
in	*delete_func*	Function pointer for the kernel deletion function
in	*control_func*	Function pointer for the kernel control function
in	*priv_arg*	Private argument passed to the kernel

Returns

A target kernel reference.

Requirement [REQ-USERKERNEL-10]: vxAddTargetKernelByName

3.1.4.5 vxRemoveTargetKernel()

```
vx_status vxRemoveTargetKernel (
            vx_target_kernel target_kernel)
```

Allows users to remove a user kernel.

Parameters

in	*target_kernel*	Handle to the target kernel to be removed

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors; any other value indicates failure.

Requirement [REQ-USERKERNEL-11]: vxRemoveTargetKernel

3.1.4.6 vxRemoveTargetKernelByName()

```
vx_status vxRemoveTargetKernelByName (
            const vx_char * kernel_name,
            const vx_char * target_name)
```

Allows users to remove a user kernel from a specific target by providing a kernel name and a target name.

Parameters

in	*kernel_name*	Name of the target kernel
in	*target_name*	Name of the target

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors; any other value indicates failure.

Requirement [REQ-USERKERNEL-12]: vxRemoveTargetKernelByName

3.1.4.7 vxSetTargetKernelInstanceContext()

```
vx_status vxSetTargetKernelInstanceContext (
            vx_target_kernel_instance target_kernel_instance,
            void * kernel_context,
            vx_uint32 kernel_context_size)
```

Associate a kernel context or handle with a target kernel instance Typically set by the kernel function during the node create phase.

The kernel context is typically a buffer containing a kernel specific data structure which may include pointers to locally allocated memory and/or parameters that need to be shared between kernel callbacks.

Parameters

in	*target_kernel_instance*	Target Kernel Instance
in	*kernel_context*	Pointer to the kernel context to be set
in	*kernel_context_size*	Size of the kernel context

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors; any other value indicates failure.

Requirement [REQ-USERKERNEL-13]: vxSetTargetKernelInstanceContext

3.1.4.8 vxGetTargetKernelInstanceContext()

```
vx_status vxGetTargetKernelInstanceContext (
            vx_target_kernel_instance target_kernel_instance,
            void ** kernel_context,
            vx_uint32 * kernel_context_size)
```

Get a kernel context or handle with a target kernel instance Typically used by the kernel function during run, control, delete phases.

The kernel context is typically a buffer containing a kernel specific data structure which may include pointers to locally allocated memory and/or parameters that need to be shared between kernel callbacks.

Parameters

`in`	*target_kernel_instance*	Handle to the target kernel instance from which to retrieve the context
`out`	*kernel_context*	Pointer to the kernel context to be retrieved
`out`	*kernel_context_size*	Size of the kernel context

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors; any other value indicates failure.

Requirement [REQ-USERKERNEL-14]: vxGetTargetKernelInstanceContext

Index