The **OpenVX™** User Kernel Tiling Extension

Version 1.0 (Provisional)

Document Revision: r29628

Khronos Vision Working Group

Editor: Shorin Kyo
Editor: Thierry Lepley
Editor: Erik Rainey
Editor: Frank Brill
Editor: Andrew Garrard

Contents

Chapter 1

User Kernel Tiling

The User Kernel Tiling facility enables optimizations of the user nodes (e.g. locality of execution or parallelism) when performing computation on the image data. Modern processors have a diverse memory hierarchy that varies from relatively small but fast and expensive memory to relatively large but slow and inexpensive memory. Image data is typically too large to fit into the fast but small memory. The ability to break the image data into smaller-sized rectangular units allows for optimized computation on these smaller units with fast memory access or parallel execution of a user node on multiple image “tiles” simultaneously. The OpenVX Graph Manager possesses the knowledge about the memory hierarchy of the target and is hence in a position to break the image data into smaller units for memory optimization. Knowledge of the memory access pattern of an algorithm is key for the graph manager to enable optimizations.

As with conventional User Kernels, Tiling User Kernels will typically be loaded and executed on HLOS/CP-U-compatible targets, not on remote processors or other accelerators — the intent of this extension is to allow efficient scheduling and memory transfer between accelerators and user functions, not to provide code to run on an accelerator, since this capability is necessarily vendor-specific. This specification does not mandate what constitutes compatible platforms.

1.1 Purpose

The purpose of having User Kernel Tiling Functions is to:

- Provide a mechanism for user-kernel programmers to communicate enough information about their computation to enable OpenVX implementations to perform some optimizations that take advantage of locality in the user-node computation, and enable parallelism of execution across local data sets.
- Define a portable method to enable tiling on compatible platforms.
- Define a simple way to enable users to write image processing functions.

1.2 Basic Design

The user kernel tiling facility is largely composed of two components:

- User Node informs Graph Manager — The author of the tilable user kernel informs the graph manager about the data requirements of the algorithm implemented by the kernel. This is primarily done by setting attributes on the kernel (for all nodes of this type), or on the node (for that particular node). This is usually done at kernel definition time (for kernel attributes) or at node creation time (for node attributes). Only single-plane images are supported as input or output arguments to tiled user nodes in this version of the specification.
- Graph Manager informs User Node — The graph manager informs the user kernel code about the rectangular sub-image, or “tile”, on which the user kernel code will run. This is primarily done via macros that the user kernel code can call to get this information at run time for a particular invocation of the user code. In this version of the tiling extension, the user functions are expected to generate the same output pixels in each output image, and to rely on the corresponding pixels of the input image, potentially with the addition of some nearby “neighborhood” pixels, to perform this calculation. The effect of a tiling user kernel that writes to its input images is implementation-defined.

1.3 Two Kernel Functions

The creator of the tiled user kernel must provide a kernel function to perform the actual computation. Optionally, the kernel creator may provide a second function with different edge-condition checks.

- One of the functions is referred to as the “flexible” function, and it may be called with irregular tile sizes and near the edge of the tile’s parent image, so it must fully check all edge conditions to handle such issues.
- The second function is referred to as the “fast” function, and it will only be called with tile sizes and pixel alignments that are multiples of the user-specified “tile block size”, and will not be called in a location that would cause references to the neighborhood region to fall outside the image.

If the kernel creator only provides the fast function, some of the output image (near the edges) will not be computed if the fast function cannot be safely called at the edge of these images. If the kernel creator only provides the flexible function, it will be called at all appropriate locations of the image, including near the edges. The OpenVX implementation should still attempt to call the flexible function with regular tile sizes and alignments where possible. If both functions are provided, the fast function should be used as much as possible given its restrictions, and the flexible function will be called elsewhere.

1.4 Glossary

Terms and Definitions of the Extension:

- Tile — The rectangle of destination pixels to be computed by the user function, whose size is less than or equal to the rectangle of the entire image. This value is determined by the OpenVX runtime. Every tile that is provided to the user node will be fully contained within the image, and no pixel of the image will be included in more than one tile. The user cannot rely on the order or concurrency of tile processing.
- Output tile block — The output tile block size imposes a coordinate grid over the image that is aligned with the top-left corner of the output image. The fast function will always be called on a tile that is aligned to this grid, and the dimensions of this tile are integer multiples of the tile block size. That is, the fast function is invoked on a tile consisting of a whole number of tile blocks. There are no restrictions on the size and alignment of a tile given to the flexible function. It is expected that the tile block size be small — potentially a single pixel — in order to maximize opportunities to use a “fast” function. The tile size should generally be much larger, and chosen by the graph manager in an implementation-specific way to make good use of cache sizes and memory transfer and to limit function call overhead, while still offering the potential for reduced memory footprint and parallel execution.
- Input tile block — The input tile block defines the dimension of a tile block in the input image corresponding to an output tile block. In OpenVX 1.0, this value is unconditionally regarded to be same as the output tile block size.
- Fast function — A function provided by the user which will be invoked only with tiles whose coordinates are aligned to, and whose dimensions are integer multiples of, the tile block size, and which will never attempt to access pixels that lie beyond the bounds of the input images (due to neighborhood). The expectation is that the user may be able to provide a more efficient implementation if the function is known to be invoked on coordinate-aligned data. For this reason, the implementation may be expected to invoke the fast function preferentially over the flexible function. If only the fast function is provided and the flexible function is not, the regions of the image which do not meet the requirements of running the fast function will not be processed. Note that coordinate alignment does not imply memory alignment, which is determined in part by image stride and is implementation-defined, though many implementations will define a relationship between coordinate and memory alignment. There is no guarantee of evaluation order of the functions, and on some implementations multiple invocations may occur concurrently. The fast function is a function pointer of type `vx_tiling_kernel_f`.
- Flexible function — A function provided by the user which can be used to process tiles that are not constrained by tile block size, and — with `VX_BORDER_MODE_SELF` — which may rely on a neighborhood that corresponds to pixels beyond the bounds of the input images. The user function should be capable of running at arbitrary coordinate alignment. If only the flexible function is provided, this will be invoked on tile-block aligned tiles as well as unaligned tiles. The user can expect aligned regions to be invoked preferentially even when only a flexible function is provided. At least one of the fast and flexible functions must be provided. The flexible function is a function pointer of type `vx_tiling_kernel_f`.

- Input neighborhood — The additional pixels surrounding the tile in the input images that are required in order to compute the output tile. The top, bottom, left and right values correspond to the offset of the edges of the neighborhood defined relative to the tile being processed. For example, a Sobel kernel requiring 3 by 3 pixels centered on the output would have an input neighborhood of (-1,1,-1,1). Functions which operate on only a single pixel have an input neighborhood of (0,0,0,0).
- Output neighborhood — The additional (previously computed) pixels surrounding the tile in the output image needed to compute the output tile. In OpenVX 1.0, this value is unconditionally regarded as zero in all four directions (top, bottom, left, right).
- Border mode — Defines the behavior of the OpenVX runtime against the determination of tiles near the image edge for non-zero neighborhood sizes. If the border mode is `VX_BORDER_MODE_SELF`, the flexible function will be invoked on tiles near the image border that would require neighborhood access beyond the image bounds, and this function is responsible for accessing any pixels whose coordinates are invalid. If the border mode is `VX_BORDER_MODE_UNDEFINED`, the graph manager will ensure that the user functions are not invoked on coordinates that, because of neighborhood, require access outside the image area.
- Tile memory — A facility for per-tile memory allocation. The amount allocated will be the amount requested via the `VX_KERNEL_ATTRIBUTE_TILE_MEMORY_SIZE` attribute (which may be zero) multiplied by the number of tile blocks in the tile being processed, and the allocation will be passed to the user-provided functions through the `tile_memory` parameter. The size allocated for that function invocation is provided via the `tile_memory_size` parameter. The valid lifetime of the given pointer is per invocation of the user function that processes a tile. This allocation is provided *per tile* — and therefore two parallel invocations of the user tiling functions will receive different allocations. This is distinct from the per-node memory managed via the `VX_KERNEL_ATTRIBUTE_LOCAL_DATA_SIZE` and `VX_KERNEL_ATTRIBUTE_LOCAL_DATA_PTR` kernel attributes.

1.5 Summary of user function calls

The following tables describe the circumstances under which each user function will be invoked, categorized by the border mode settings of the node.

1.5.1 User function invocation according to which functions are defined

Functions provided	**VX_BORDER_MODE_UNDEFINED**	**VX_BORDER_MODE_SELF**
If only the FAST function is defined...	The Fast function is invoked on the largest aligned image subset that avoids out-of-bounds accesses. Other pixels are not processed.	Prohibited (graph validation fails).
If only the FLEXIBLE function is defined...	The flexible function is invoked on the largest image subset that ensures neighborhood accesses are within bounds, irrespective of alignment. Other pixels are not processed.	The flexible function is invoked for all pixels in the output image irrespective of alignment, including pixels that correspond to neighborhood accesses beyond the image bounds.
If both the FAST and FLEXIBLE functions are defined...	The fast function is invoked on the largest aligned image subset that avoids out-of-bounds neighborhood accesses. The flexible function is invoked for the remaining pixels that do not require out-of-bounds accesses.	The fast function is invoked on the largest aligned image subset that avoids out-of-bounds neighborhood accesses. The flexible function is invoked on remaining pixels.

Note that a missing flexible function when using `VX_BORDER_MODE_SELF` is treated as an error even if the fast function might never be expected to be called (for example, if the image size is divisible by the tile block size and the neighborhood is 0).

1.5.2 User function invocation according to which pixels are being processed

Pixels to process	VX_BORDER_MODE_UNDEFINED	VX_BORDER_MODE_SELF
Pixels that require access beyond the image bounds (due to neighborhood)	Not processed by any function	Processed by the flexible function, which must be provided
Pixels that require access only within the image, but in an unaligned tile	Processed by the flexible function (if provided) and not processed otherwise	Processed by the flexible function, which must be provided
Pixels within an aligned tile that do not access beyond the image bounds	Processed by the fast function if provided, otherwise by the flexible function	Processed by the fast function if provided, otherwise by the flexible function

1.6 Adding a Tiling Kernel Function

```
    vx_status status = VX_SUCCESS;
    vx_uint32 k = 0;
    for (k = 0; k < dimof(tiling_kernels); k++)
    {
        vx_kernel kernel = vxAddTilingKernel(context,
                tiling_kernels[k].name,
                tiling_kernels[k].enumeration,
                tiling_kernels[k].flexible_function,
                tiling_kernels[k].fast_function,
                tiling_kernels[k].num_params,
                tiling_kernels[k].input_validator,
                tiling_kernels[k].output_validator);
        if (kernel)
        {
            vx_uint32 p = 0;
            for (p = 0; p < tiling_kernels[k].num_params; p++)
            {
                status |= vxAddParameterToKernel(kernel, p,
                            tiling_kernels[k].parameters[p].direction,
                            tiling_kernels[k].parameters[p].data_type,
                            tiling_kernels[k].parameters[p].state);
            }
            status |= vxSetKernelAttribute(kernel,
      VX_KERNEL_ATTRIBUTE_INPUT_NEIGHBORHOOD, &tiling_kernels[k].nbhd,
      sizeof(vx_neighborhood_size_t));
            status |= vxSetKernelAttribute(kernel,
      VX_KERNEL_ATTRIBUTE_OUTPUT_TILE_BLOCK_SIZE, &tiling_kernels[k].
      block, sizeof(vx_tile_block_size_t));
            status |= vxSetKernelAttribute(kernel, VX_KERNEL_ATTRIBUTE_BORDER, &
      tiling_kernels[k].border, sizeof(vx_border_mode_t));
            if (status != VX_SUCCESS)
            {
                vxRemoveKernel(kernel);
            }
            else
            {
                status = vxFinalizeKernel(kernel);
            }
            if (status != VX_SUCCESS)
            {
                printf("Failed to publish kernel %s\n", tiling_kernels[k].name);
                break;
            }
        }
    }
```

1.7 Example Kernels

Users can add simple filter-style kernels similar to this example of a Gaussian Blur.

```
void gaussian_image_tiling_fast(void * VX_RESTRICT parameters[VX_RESTRICT],
                                void * VX_RESTRICT tile_memory,
                                vx_size tile_memory_size)
{
    vx_uint32 x, y;
    vx_tile_t *in = (vx_tile_t *)parameters[0];
    vx_tile_t *out = (vx_tile_t *)parameters[1];

    for (y = 0; y < vxTileHeight(out, 0); y += vxTileBlockHeight(out))
    {
        for (x = 0; x < vxTileWidth(out, 0); x += vxTileBlockWidth(out))
        {
            vx_uint32 sum = 0;
            /* this math covers a {-1,1},{-1,1} neighborhood and a block of 1x1.
             * an internal set of for loops could have been placed here instead.
             */
            sum += vxImagePixel(vx_uint8, in, 0, x, y, -1, -1);
            sum += vxImagePixel(vx_uint8, in, 0, x, y,  0, -1) << 1;
            sum += vxImagePixel(vx_uint8, in, 0, x, y, +1, -1);
            sum += vxImagePixel(vx_uint8, in, 0, x, y, -1,  0) << 1;
            sum += vxImagePixel(vx_uint8, in, 0, x, y,  0,  0) << 2;
            sum += vxImagePixel(vx_uint8, in, 0, x, y, +1,  0) << 1;
            sum += vxImagePixel(vx_uint8, in, 0, x, y, -1, +1);
            sum += vxImagePixel(vx_uint8, in, 0, x, y,  0, +1) << 1;
            sum += vxImagePixel(vx_uint8, in, 0, x, y, +1, +1);
            sum >>= 4;
            if (sum > 255)
                sum = 255;
            vxImagePixel(vx_uint8, out, 0, x, y, 0, 0) = (vx_uint8)sum;
        }
    }
}
```

Note that this function has a non-zero neighborhood, and performs no special checking for running at the image borders. Therefore it must be used as a “fast” function and, unless a corresponding “flexible” function is provided, must be used with VX_BORDER_MODE_UNDEFINED.

Users can also add more complex addressing functionality in order to optimize their code in this example which uses two inputs:

```
void add_image_tiling(void * VX_RESTRICT parameters[VX_RESTRICT],
                      void * VX_RESTRICT tile_memory,
                      vx_size tile_memory_size)
{
    vx_uint32 i,j;
    vx_tile_t *in0 = (vx_tile_t *)parameters[0];
    vx_tile_t *in1 = (vx_tile_t *)parameters[1];
    vx_tile_t *out = (vx_tile_t *)parameters[2];

    for (j = 0u; j < vxTileHeight(out,0); j+=vxTileBlockHeight(out))
    {
        for (i = 0u; i < vxTileWidth(out,0); i+=vxTileBlockWidth(out))
        {
            /* this math covers a 1x1 block and has no neighborhood */
            vx_uint16 pixel = vxImagePixel(vx_uint8, in0, 0, i, j, 0, 0) +
                              vxImagePixel(vx_uint8, in1, 0, i, j, 0, 0);
            if (pixel > INT16_MAX)
                pixel = INT16_MAX;
            vxImagePixel(vx_int16, out, 0, i, j, 0, 0) = (vx_int16)pixel;
        }
    }
}
```

With a tile block width and height of one, combined with the zero neighborhood, this function can be used as a “flexible” function. With a larger tile block size, this is a “fast” function.

For better performance, the user might split this into two functions. Firstly, a “fast” function:

```
void add_image_tiling_fast(void * VX_RESTRICT parameters[VX_RESTRICT],
                           void * VX_RESTRICT tile_memory,
                           vx_size tile_memory_size)
{
    vx_uint32 i,j,ii,jj;
    vx_tile_t *in0 = (vx_tile_t *)parameters[0];
    vx_tile_t *in1 = (vx_tile_t *)parameters[1];
    vx_tile_t *out = (vx_tile_t *)parameters[2];

    for (j = 0u; j < vxTileHeight(out,0); j+=16)
    {
        for (jj = 0u; jj < 16u; ++jj)
        {
            for (i = 0u; i < vxTileWidth(out,0); i+=16)
            {
                for (ii = 0u; ii < 16u; ++ii) {
                    /* this math covers a 1x1 block and has no neighborhood */
                    vx_uint16 pixel = vxImagePixel(vx_uint8, in0, 0, i+ii, j+jj, 0, 0) +
                        vxImagePixel(vx_uint8, in1, 0, i+ii, j+jj, 0, 0);
                    if (pixel > INT16_MAX)
                        pixel = INT16_MAX;
                    vxImagePixel(vx_int16, out, 0, i+ii, j+jj, 0, 0) = (vx_int16)pixel;
                }
            }
        }
    }
}
```

This fast function has a fixed 16 by 16 tile block size, the loops of which the compiler may be able to unroll or otherwise optimize — the pixel access macro is designed to give the compiler the best chance of doing this. Less portable code may be able to rely on byte alignment and fast memory accesses, and perform SIMD operations in the fast function to maximize the host CPU performance.

In addition to performance optimization, many algorithms have tile block size requirements due to spatial effects on pixels. For example, ordered dithering, UV upsampling and debayering all require pixels to be processed differently according to location within a repeating block.

This would be complemented by a corresponding "flexible" function:

```
void add_image_tiling_flexible(void * VX_RESTRICT parameters[VX_RESTRICT],
                               void * VX_RESTRICT tile_memory,
                               vx_size tile_memory_size)
{
    vx_uint32 i,j;
    vx_tile_t *in0 = (vx_tile_t *)parameters[0];
    vx_tile_t *in1 = (vx_tile_t *)parameters[1];
    vx_tile_t *out = (vx_tile_t *)parameters[2];

    for (j = 0u; j < vxTileHeight(out,0); ++j)
    {
        for (i = 0u; i < vxTileWidth(out,0); ++i)
        {
            /* this math covers a 1x1 block and has no neighborhood */
            vx_uint16 pixel = vxImagePixel(vx_uint8, in0, 0, i, j, 0, 0) +
                              vxImagePixel(vx_uint8, in1, 0, i, j, 0, 0);
            if (pixel > INT16_MAX)
                pixel = INT16_MAX;
            vxImagePixel(vx_int16, out, 0, i, j, 0, 0) = (vx_int16)pixel;
        }
    }
}
```

This function ignores the tile block and processes the tile given to it one pixel at a time. It is expected that the flexible function should be called for much less of the image than the fast function, so this overhead should be considered acceptable. This applies whether the flexible function is invoked because of tile alignment or in order to support out-of-bounds accesses, and users writing portable OpenVX code should be able to devote effort to the fast function. However, it is legal for an implementation with minimal tiling support to process an entire image in a single, possibly unaligned, tile for portability reasons.

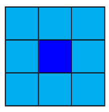
1.8 Example diagrams

1.8.1 A trivial kernel

The simplest user functions (that depend on any inputs at all) will read only the pixel in the input image(s) that correspond(s) to the pixel in the output image(s). Such a function requires a neighborhood size of 0 in each direction. If only a single image is processed at a time, the tile block size should be set to 1 by 1. Here we show the tile block — and therefore minimum tile size — in blue (trivially).

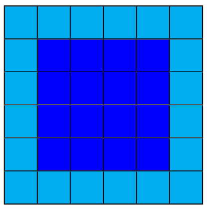
1.8.2 A user function with a neighborhood

A more complex user function may require source pixels surrounding the coordinate of the output pixel in order to work. For example, a Sobel filter requires the a three by three block centered on the current pixel. Here we show the neighborhood in cyan around the single pixel to be processed, in blue — each square represents one pixel. This neighborhood would be specified as "-1,1,-1,1".



1.8.3 A faster function with a neighborhood

The user may choose to optimize the same function by making it work on blocks of 4 by 4 pixels, rather than one pixel at a time. In this case, the tile block should be set to 4 by 4. Depending on the implementation, the neighborhood may be unmodified. If a fast function is defined that uses this 4 by 4 tile block, the user should still offer a flexible function that can process tiles that are not aligned to this 4 by 4 grid, since otherwise some portions of the image area may be left unprocessed. Here we show the 4 by 4 tile block with a single pixel neighborhood, specified in the same way as before.



Tiles processed using the "fast" function will be aligned to a 4x4 grid, and some multiple of this tile block size; the fast function will be able to access valid pixels from the neighborhood area (which will surround the complete tile by a single pixel).

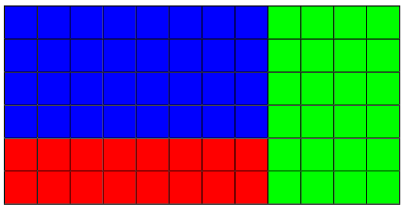
1.8.4 Tiles and tile blocks

Consider a simple function that has no neighborhood - let us say a function which posterizes the image by replicating the top four bits of an 8-bit input into the lower four bits:

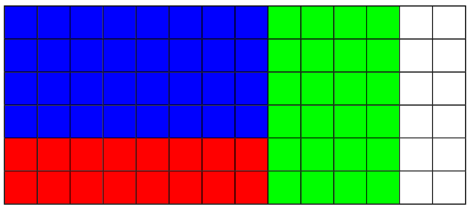
```
out(x,y) = (in(x,y) & 0xF0) | (in(x,y) >> 4)
```

If the implementation guarantees 32-bit alignment for the start of each row — *not* something guaranteed by OpenVX, but likely common — then the user could write a "fast" function which processes four pixels at once, using a 32-bit integer. This function would then have a 4 by 1 "tile block".

In processing a 12 by 6 image using this function, the graph manager might choose to break the image into three rectangular "tiles". An example in shown in different colors below. The fast function can be called for each of these tiles. It is up to a graph manager to decide different arrangements, and to decide the order in which the fast function would be invoked for these tiles — including whether the tiles are evaluated in parallel — but the tile block size guarantees that the tiles will be multiples of 4 by 1 pixels.



Next, consider what happens if the image is 14 by 6 pixels:

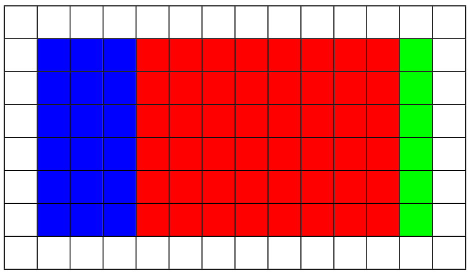


In this case, the 2-by-6 strip at the right edge of the image cannot be processed by tile-block-aligned tiles, and therefore the fast function cannot process these pixels. If the user has provided a "flexible" function, that function will be used to process these pixels (still provided with rectangular tiles of data).

1.8.5 Example of borders

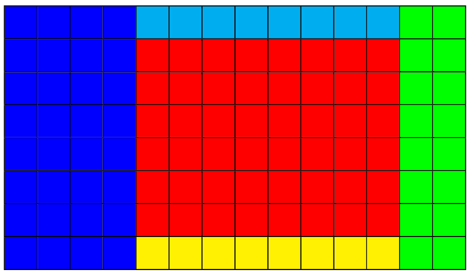
We might imagine a function that is similar to the above, but which requires access to all the source pixels surrounding the coordinate of the destination pixel in order to operate. While it may process a single pixel, the user may choose to perform a similar optimization to that discussed in the previous section, resulting in a 4-by-1 tile block with a one-pixel border neighborhood around it.

With the user node's border mode set to `VX_BORDER_MODE_UNDEFINED`, the graph manager might choose to break the same 14 by 8 image into the following tiles for the user functions to process:



In this case, the white pixels at the image edge are not processed at all. The red tile is aligned to tile blocks, and can be processed with a fast function, if one is provided (otherwise the flexible function is used). The blue and green tiles do not start on (blue tile) or end on (green tile) a 4 by 1 tile block grid, so these pixels must be processed by the flexible function. If no flexible function is provided, these pixels will not be processed.

With the user node's border mode set to `VX_BORDER_MODE_SELF`, the graph manager might choose to break the same 14 by 8 image into the following tiles for the user functions to process:



With `VX_BORDER_MODE_SELF`, the flexible function *must* be provided. In this example, only the red tile can be processed by the fast function (if provided), since only this tile is aligned to a tile block boundary and has no requirements on pixels beyond the image borders. The blue, cyan and yellow regions are aligned to the tile block boundary, but must be processed by the flexible function because the neighborhood of these pixels extends beyond the image boundary. The green tile is additionally not aligned to the tile block size.

1.9 Informative Coding Guidelines

Tiled user kernel functions should be coded in a very specific manner to maintain the requirement of *data independence* in that all blocks should be computed in a *functionally-independent* manner. This allows the implementation greater flexibility in optimization choices. These informal requirements include:

- Avoid using global shared memory.
- Avoid using blocking access protected mechanisms (semaphores, etc).
- Avoid using dynamic memory allocation functions at runtime.
- Avoid calling other functions internally in the tile function.
- Do try to write the your loops in an efficient manner for your compiler environment.

Chapter 2

Module Documentation

2.1 Extension: User Tiling API

The Khronos Extension for User Tiling Functions.

Data Structures

- struct vx_image_description_t

  A structure which describes the tile's parent image. More...
- struct vx_neighborhood_size_t

  The User Tiling Function Neighborhood declaration. More...
- struct vx_tile_block_size_t

  The User Tiling Function tile block size declaration. More...
- struct vx_tile_t

  The tile structure declaration. More...

Macros

- #define VX_MAX_TILING_PLANES (4)

  The maximum number of planes in a tiled image.
- #define VX_RESTRICT restrict

  A platform wrapper for the restrict keyword.
- #define vxImageHeight(ptile) ((ptile))->image.height)

  The full height of the tile's parent image in pixels.
- #define vxImageOffset(ptile, i, x, y, ox, oy)

  Computes the offset within an image.
- #define vxImagePixel(type, ptile, i, x, y, ox, oy) ∗((type ∗)(&((vx_uint8 ∗)(ptile)->base[i])[vxImageOffset(ptile, i, x, y, ox, oy)]))

  Accesses an image pixel as a type-cast indexed pointer dereference.
- #define vxImageWidth(ptile) ((ptile))->image.width)

  The full width of the tile's parent image in pixels.
- #define vxNeighborhoodBottom(ptile) ((ptile)->neighborhood.bottom)

  The simple wrapper to access each image's neighborhood +Y value.
- #define vxNeighborhoodLeft(ptile) ((ptile)->neighborhood.left)

  The simple wrapper to access each image's neighborhood -X value.
- #define vxNeighborhoodRight(ptile) ((ptile)->neighborhood.right)

  The simple wrapper to access each image's neighborhood +X value.
- #define vxNeighborhoodTop(ptile) ((ptile)->neighborhood.top)

  The simple wrapper to access each image's neighborhood -Y value.
- #define vxTileBlockHeight(ptile) ((ptile)->tile_block.height)

The tile block height.

- #define vxTileBlockWidth(ptile) ((ptile)->tile_block.width)

  The tile block width.

- #define vxTileHeight(ptile, index) ((ptile)->addr[index].dim_y)

  The height of the tile in pixels.

- #define vxTileWidth(ptile, index) ((ptile)->addr[index].dim_x)

  The width of the tile in pixels.

- #define vxTileX(ptile) ((ptile)->tile_x)

  The offset between the left edge of the image and the left edge of the tile, in pixels.

- #define vxTileY(ptile) ((ptile)->tile_y)

  The offset between the top edge of the image and the top edge of the tile, in pixels.

Typedefs

- typedef void(∗ vx_tiling_kernel_f)(void ∗restrict parameters[restrict], void ∗restrict tile_memory, vx_size tile_-memory_size)

  Tiling Kernel function typedef for User Tiling Kernels.

Enumerations

- enum vx_border_mode_tiling_e { VX_BORDER_MODE_SELF = (((VX_ID_KHRONOS) << 20) | (VX_ENUM-_BORDER_MODE << 12)) + 0x3 }

  The tiling border mode extensions.

- enum vx_kernel_attribute_tiling_e {
  VX_KERNEL_ATTRIBUTE_INPUT_NEIGHBORHOOD = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERN-EL << 8)) + 0x7,
  VX_KERNEL_ATTRIBUTE_OUTPUT_TILE_BLOCK_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KE-RNEL << 8)) + 0x8,
  VX_KERNEL_ATTRIBUTE_BORDER = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x9,
  VX_KERNEL_ATTRIBUTE_TILE_MEMORY_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0xA }

  The User Kernel Tiling Attributes.

- enum vx_node_attribute_tiling_e {
  VX_NODE_ATTRIBUTE_INPUT_NEIGHBORHOOD = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x7,
  VX_NODE_ATTRIBUTE_OUTPUT_TILE_BLOCK_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x8,
  VX_NODE_ATTRIBUTE_TILE_MEMORY_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0xA }

  The User Node Tiling Attributes.

Functions

- vx_kernel vxAddTilingKernel (vx_context context, vx_char name[(256)], vx_enum enumeration, vx_tiling_kernel-_f flexible_func_ptr, vx_tiling_kernel_f fast_func_ptr, vx_uint32 num_params, vx_kernel_input_validate_f input, vx_-kernel_output_validate_f output)

  Allows a user to add a tile-able kernel to the OpenVX system.

2.1.1 Detailed Description

The Khronos Extension for User Tiling Functions.

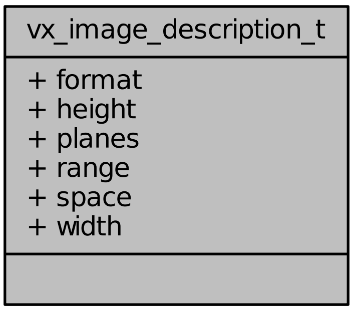
2.1.2 Data Structure Documentation

struct vx_image_description_t

A structure which describes the tile's parent image.

Definition at line 86 of file vx_khr_tiling.h.

Collaboration diagram for vx_image_description_t:



Data Fields

vx_df_image	format	The `vx_df_image_e` of the image.
vx_uint32	height	Height of the image.
vx_uint32	planes	The number of planes in the image.
vx_enum	range	The `vx_channel_range_e` enumeration.
vx_enum	space	The `vx_color_space_e` enumeration.
vx_uint32	width	Width of the image.

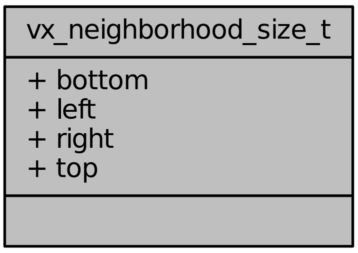
struct vx_neighborhood_size_t

The User Tiling Function Neighborhood declaration.

The author of a User Tiling Kernel will use this structure to define the neighborhood surrounding the tile block.

Definition at line 76 of file vx_khr_tiling.h.

Collaboration diagram for vx_neighborhood_size_t:

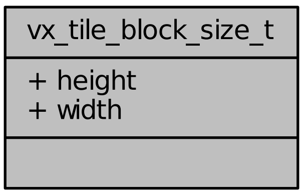


Data Fields

vx_int32	bottom	Bottom of the tile block.
vx_int32	left	Left of the tile block.
vx_int32	right	Right of the tile block.
vx_int32	top	Top of the tile block.

struct vx_tile_block_size_t

The User Tiling Function tile block size declaration.
The author of a User Tiling Kernel will use this structure to define the dimensionality of the tile block.
Definition at line 66 of file vx_khr_tiling.h.
Collaboration diagram for vx_tile_block_size_t:



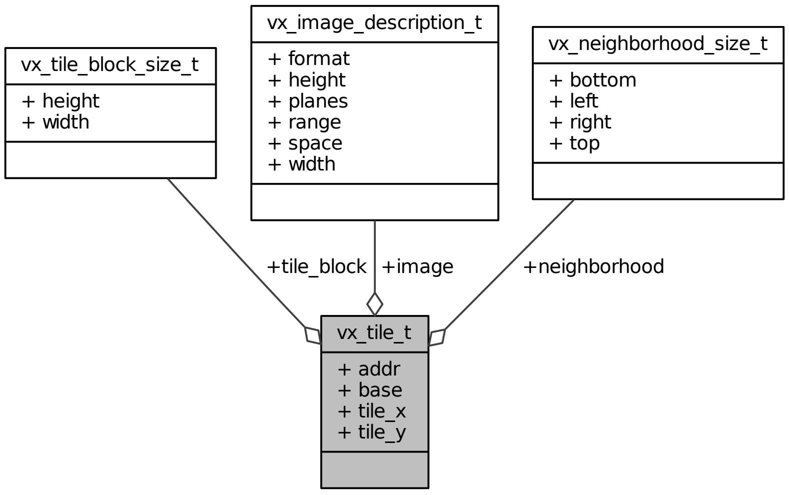
Data Fields

vx_int32	height	Tile block height in pixels.
vx_int32	width	Tile block width in pixels.

struct vx_tile_t

The tile structure declaration.
Definition at line 103 of file vx_khr_tiling.h.

Collaboration diagram for vx_tile_t:



Data Fields

vx_imagepatch_addressing_t	addr[(4)]	The array of addressing structure to describe each plane.
vx_uint8 *restrict	base[(4)]	The array of pointers to the tile's image plane.
vx_image_description_t	image	The description and attributes of the image.
vx_neighborhood_size_t	neighborhood	The neighborhood definition.
vx_tile_block_size_t	tile_block	The output block size structure.
vx_uint32	tile_x	The top left X pixel index within the width dimension of the image.
vx_uint32	tile_y	The top left Y pixel index within the height dimension of the image.

2.1.3 Macro Definition Documentation

#define vxImageHeight(*ptile*) ((ptile))->image.height)

The full height of the tile's parent image in pixels.

Parameters

in	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 127 of file vx_khr_tiling.h.

#define vxImageOffset(*ptile, i, x, y, ox, oy*)

Value:

```
((ptile)->addr[i].stride_y * (vx_int32)(((vx_int32)((oy)+(y)) * (vx_int32)(ptile)->addr[i].scale_y)/(
       vx_int32)VX_SCALE_UNITY)) + \
         ((ptile)->addr[i].stride_x * (vx_int32)(((vx_int32)((ox)+(x)) * (vx_int32)(ptile)->addr[i].scale_x)
       /(vx_int32)VX_SCALE_UNITY))
```

Computes the offset within an image.

Parameters

`in`	*ptile*	The pointer to the vx_tile_t structure.
`in`	*i*	The plane index.
`in`	*x*	The Width Coordinates.
`in`	*y*	The Height Coordinates.
`in`	*ox*	The X offset.
`in`	*oy*	The Y offset.

Definition at line 322 of file vx_khr_tiling.h.

#define vxImagePixel(*type, ptile, i, x, y, ox, oy*) ∗((type ∗)(&((vx_uint8 ∗)(ptile)->base[i])[vxImageOffset(ptile, i, x, y, ox, oy)]))

Accesses an image pixel as a type-cast indexed pointer dereference.

Parameters

`in`	*type*	The type of the image pixel. Example values are `vx_uint8`, `vx_uint16`, `vx_uint32`, etc.
`in`	*ptile*	The pointer to the vx_tile_t structure.
`in`	*i*	The plane index.
`in`	*x*	The Center Pixel in Width Coordinates.
`in`	*y*	The Center Pixel in Height Coordinates.
`in`	*ox*	The X offset.
`in`	*oy*	The Y offset.

Definition at line 338 of file vx_khr_tiling.h.

#define vxImageWidth(*ptile*) ((ptile))->image.width)

The full width of the tile's parent image in pixels.

Parameters

`in`	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 134 of file vx_khr_tiling.h.

#define vxNeighborhoodBottom(*ptile*) ((ptile)->neighborhood.bottom)

The simple wrapper to access each image's neighborhood +Y value.

Parameters

`in`	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 206 of file vx_khr_tiling.h.

#define vxNeighborhoodLeft(*ptile*) ((ptile)->neighborhood.left)

The simple wrapper to access each image's neighborhood -X value.

Parameters

`in`	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 185 of file vx_khr_tiling.h.

#define vxNeighborhoodRight(*ptile*) ((ptile)->neighborhood.right)

The simple wrapper to access each image's neighborhood +X value.

Parameters

in	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 192 of file vx_khr_tiling.h.

#define vxNeighborhoodTop(*ptile*) ((ptile)->neighborhood.top)

The simple wrapper to access each image's neighborhood -Y value.

Parameters

in	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 199 of file vx_khr_tiling.h.

#define vxTileBlockHeight(*ptile*) ((ptile)->tile_block.height)

The tile block height.

Parameters

in	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 171 of file vx_khr_tiling.h.

#define vxTileBlockWidth(*ptile*) ((ptile)->tile_block.width)

The tile block width.

Parameters

in	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 178 of file vx_khr_tiling.h.

#define vxTileHeight(*ptile, index*) ((ptile)->addr[index].dim_y)

The height of the tile in pixels.

Parameters

in	*ptile*	The pointer to the vx_tile_t structure.
in	*index*	The plane index.

Definition at line 164 of file vx_khr_tiling.h.

#define vxTileWidth(*ptile, index*) ((ptile)->addr[index].dim_x)

The width of the tile in pixels.

Parameters

in	*ptile*	The pointer to the vx_tile_t structure.
in	*index*	The plane index.

Definition at line 156 of file vx_khr_tiling.h.

#define vxTileX(*ptile*) ((ptile)->tile_x)

The offset between the left edge of the image and the left edge of the tile, in pixels.

Parameters

in	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 141 of file vx_khr_tiling.h.

#define vxTileY(*ptile*) ((ptile)->tile_y)

The offset between the top edge of the image and the top edge of the tile, in pixels.

Parameters

`in`	*ptile*	The pointer to the vx_tile_t structure.

Definition at line 148 of file vx_khr_tiling.h.

2.1.4 Typedef Documentation

vx_tiling_kernel_f

Tiling Kernel function typedef for User Tiling Kernels.

Note

Tiles may come in any dimension and are not guaranteed to be delivered in any particular order.

Parameters

`in`	*parameters*	The array abstract pointers to parameters.
`in`	*tile_memory*	The local tile memory pointer if requested, otherwise NULL.
`in`	*tile_memory_size*	The size of the local tile memory, if not requested, 0.

Definition at line 305 of file vx_khr_tiling.h.

2.1.5 Enumeration Type Documentation

enum vx_border_mode_tiling_e

The tiling border mode extensions.

Enumerator

VX_BORDER_MODE_SELF This value indicates that the author of the tiling kernel wrote code to handle border conditions into the kernel itself. If this mode is set, it can not be overriden by a call to the vxSetNodeAttribute with VX_NODE_ATTRIBUTE_BORDER_MODE.

Definition at line 282 of file vx_khr_tiling.h.

enum vx_kernel_attribute_tiling_e

The User Kernel Tiling Attributes.

Enumerator

VX_KERNEL_ATTRIBUTE_INPUT_NEIGHBORHOOD This allows a tiling mode kernel to set its input neighborhood.

VX_KERNEL_ATTRIBUTE_OUTPUT_TILE_BLOCK_SIZE This allows a tiling mode kernel to set its output tile block size.

VX_KERNEL_ATTRIBUTE_BORDER This allows the author to set the border mode on the tiling kernel.

VX_KERNEL_ATTRIBUTE_TILE_MEMORY_SIZE This determines the per tile memory allocation.

Definition at line 243 of file vx_khr_tiling.h.

enum vx_node_attribute_tiling_e

The User Node Tiling Attributes.

Note

These are largely unusable by the tiling function, as it doesn't give you the node reference!

Enumerator

VX_NODE_ATTRIBUTE_INPUT_NEIGHBORHOOD This allows a tiling mode node to get its input neighborhood.

VX_NODE_ATTRIBUTE_OUTPUT_TILE_BLOCK_SIZE This allows a tiling mode node to get its output tile block size.

VX_NODE_ATTRIBUTE_TILE_MEMORY_SIZE This is the size of the tile local memory area.

Definition at line 264 of file vx_khr_tiling.h.

2.1.6 Function Documentation

vx_kernel vxAddTilingKernel (vx_context *context,* vx_char *name[(256)],* vx_enum *enumeration,* vx_tiling_kernel_f *flexible_func_ptr,* vx_tiling_kernel_f *fast_func_ptr,* vx_uint32 *num_params,* vx_kernel_input_validate_f *input,* vx_kernel_output_validate_f *output*)

Allows a user to add a tile-able kernel to the OpenVX system.

Parameters

`in`	*context*	The handle to the implementation context.
`in`	*name*	The string to be used to match the kernel.
`in`	*enumeration*	The enumerated value of the kernel to be used by clients.
`in`	*flexible_func_ptr*	The process-local flexible function pointer to be invoked.
`in`	*fast_func_ptr*	The process-local fast function pointer to be invoked.
`in`	*num_params*	The number of parameters for this kernel.
`in`	*input*	The pointer to a function which will validate the input parameters to this kernel.
`in`	*output*	The pointer to a function which will validate the output parameters to this kernel.

Note

Tiling Kernels do not have access to any of the normal node attributes listed in vx_node_attribute_e.

Postcondition

Call `vxAddParameterToKernel` for as many parameters as the function has, then call `vxFinalizeKernel`.

Return values

0	Indicates that an error occurred when adding the kernel. Note that the fast or flexible formula, but not both, can be NULL.

Index