Implementation of an EVE App

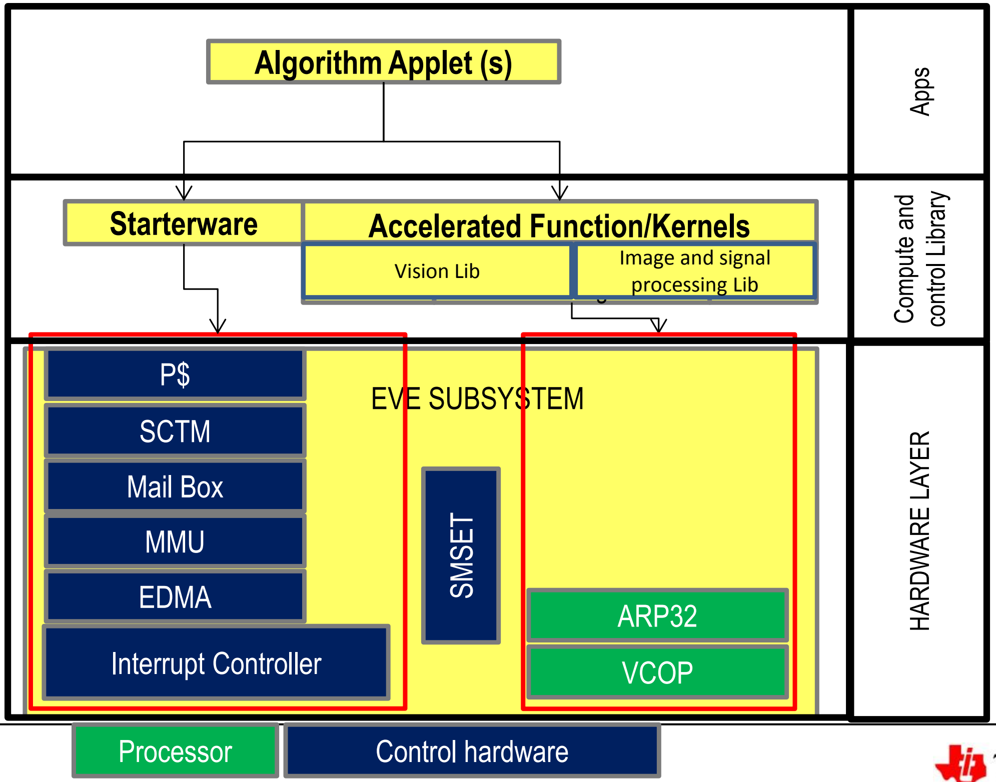
Software components

EVE software has three key components

Starterware – This component contains the APIs to program different control modules of EVE subsystem

Acclerated Functions – It is set of accelerated functions utilizing EVE Vector Co-processor (VCOP) for different applications (vision and imaging). These functions expect input and outout in EVE subsystem memory.

Apps – These are high-level applications working on the data in external memory and underneath utilizing starterware and accelerated functions. Example of such applications are resizing of an image, Harris corner detection etc.



Two different ways of developing an EVE APP

There are two ways of developing an EVE APP:

- Low-level Starterware based development
- High level graph based development

Two different ways of developing an EVE APP

There are two ways of developing an EVE APP:

- Low-level Starterware based development:
  - Rely on starterware to program the EDMA.
    - Doc: Starterware\docs\eve_starterware_userguide.pdf
    - Example code: apps\apps_nonbam
  - Little hardware abstraction, higher visibility into the basic components of EVE: EDMA, buffer switching, ping-pong buffering, memory layout.
  - Faster ramp-up time for training but less flexibility in term of code upgrade: once an algorithm is written for a specific use case, some work is required to adapt it to other use case.

Two different ways of developing an EVE APP

There are two ways of developing an EVE APP:

- High level graph based development:
  - Rely on BAM (Block acceleration manager) framework
    - Doc: algframework\docs\bam_userguide.pdf
    - Training: https://cdds.ext.ti.com/ematrix/common/emxNavigator.jsp?objectId=28670.42872.52654.60377
    - Example code: apps\
  - High hardware abstraction, little visibility into the basic components of EVE: EDMA, buffer switching, ping-pong buffering, memory layout.
  - Ramp-up time for training is ~ 2 week + wrapper functions need to be implemented. However high flexibility, easier to maintain/customize complex algorithm because of plug and play approach to create an algorithm.

Low-level starterware based development

- Driver file for testing 2-D FIR filter example:

```
apps\apps_nonbam\test\evelib_fir_filter_2d_test.c
```

- Linker command file:

```
apps\apps_nonbam\test\common\linker.cmd
```

- Implementation of 2-D FIR filter APP:

```
apps\apps_nonbam\src\evelib_fir_filter_2d.c
```

- Block-based auto-increment function:

```
apps\apps_nonbam\common\eve_algo_dma_auto_incr.c
```

- Starterware library:

```
Starterware\libs\vayu\eve\release\libevestarterware_eve.lib
```

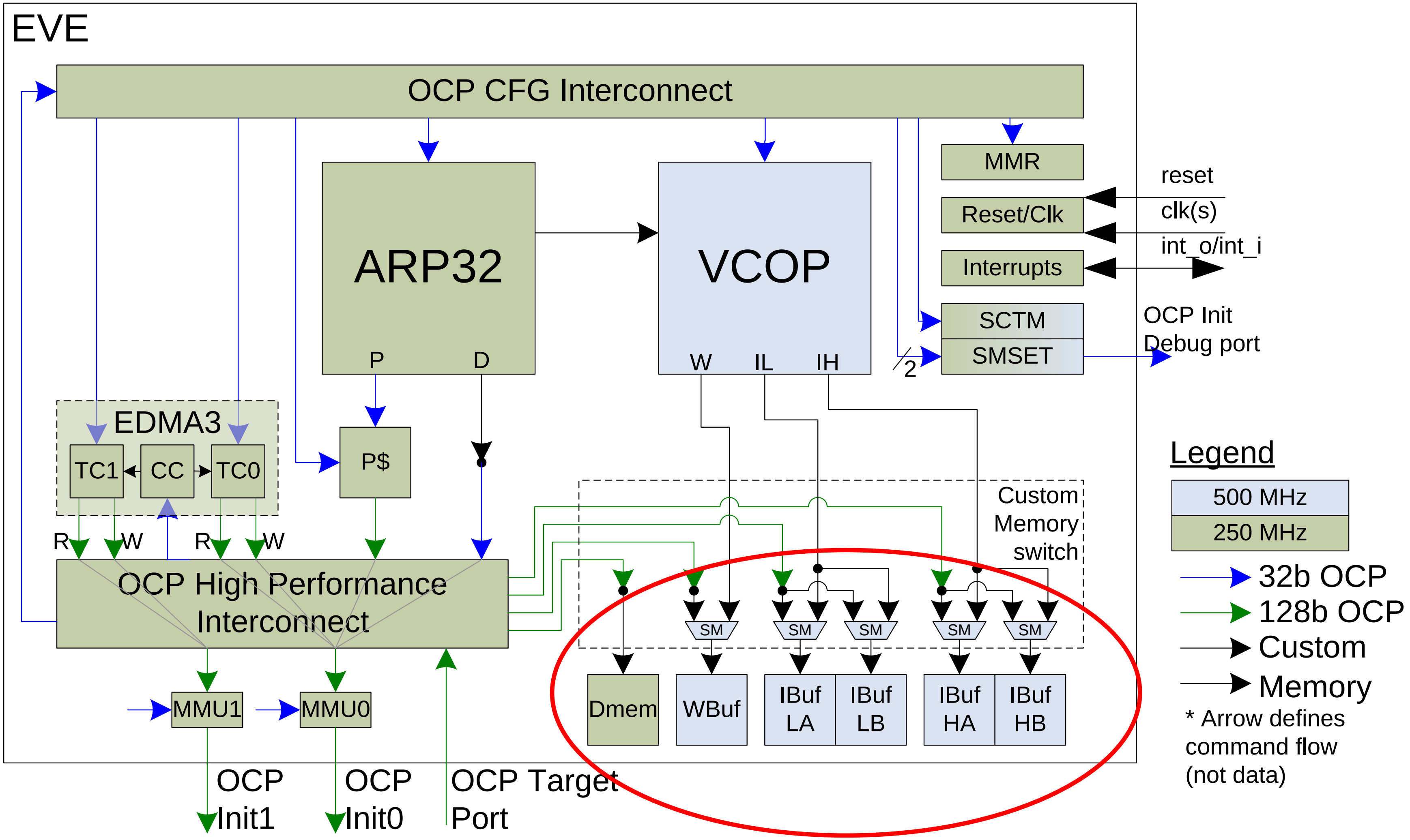
Linker cmd file – memory map

In `apps/apps_nonbam/test/common`

```
MEMORY
{
    PAGE 0:
      VECMEM  :    origin      = 0x80000000, length = 0x0100
      CMDMEM  :    origin      = 0x80000100, length = 0x1000
      EXTMEM  :    origin      = 0x80001100, length = 0x20000

    PAGE 1:
      DATMEM  :    origin = 0x40020000 length = 0x8000
      WMEM    :    origin = 0x40040000 length = 0x7FE0
      IMEMLA  :    origin = 0x40050000 length = 0x4000
      IMEMHA  :    origin = 0x40054000 length = 0x4000
      IMEMLB  :    origin = 0x40070000 length = 0x4000
      IMEMHB  :    origin = 0x40074000 length = 0x4000
      GEM0_L2_MEM: origin = 0x40800000 length = 0x8000
      EXTDMEM :    origin = 0x80030000 length = 0x2000000
      L3MEM   :    origin = 0x40300000, length = 0x100000
}
```

Memory Sections - recap



Data flow through VCOP

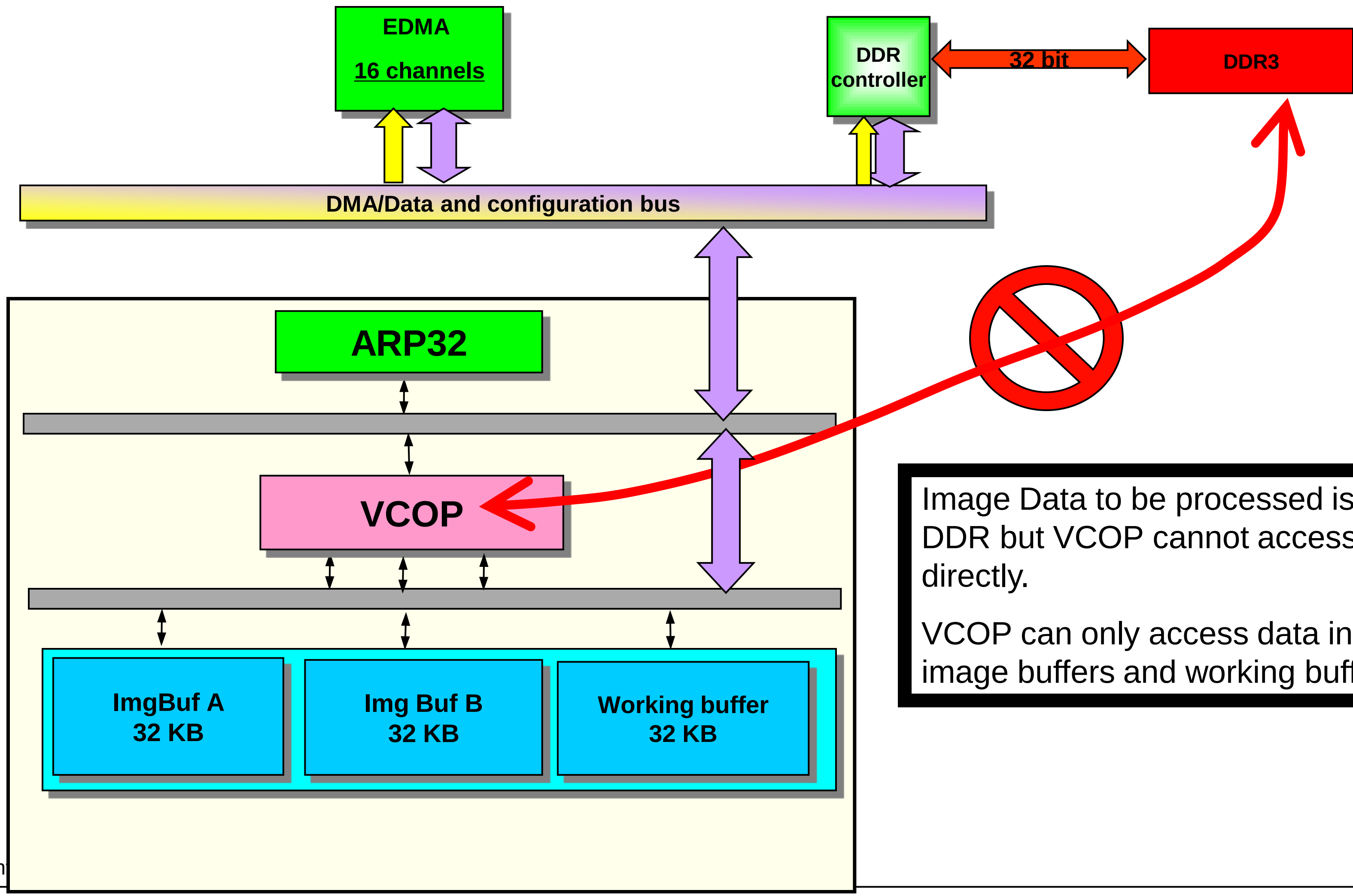
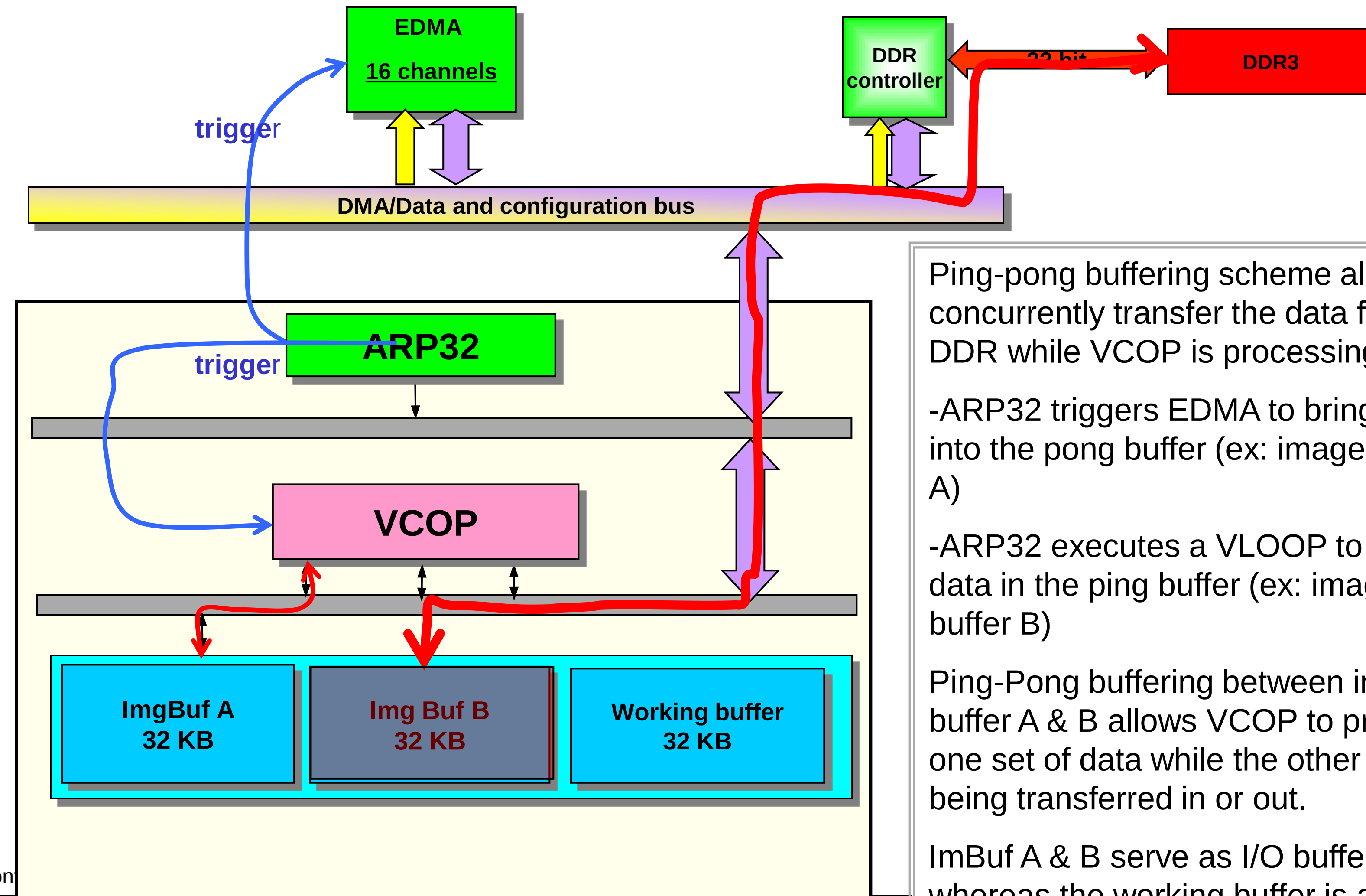


Image Data to be processed is in DDR but VCOP cannot access DDR directly.

VCOP can only access data in the image buffers and working buffer.

Data flow through VCOP



Ping-pong buffering scheme allows to concurrently transfer the data from/to DDR while VCOP is processing data:

-ARP32 triggers EDMA to bring data into the pong buffer (ex: image buffer A)

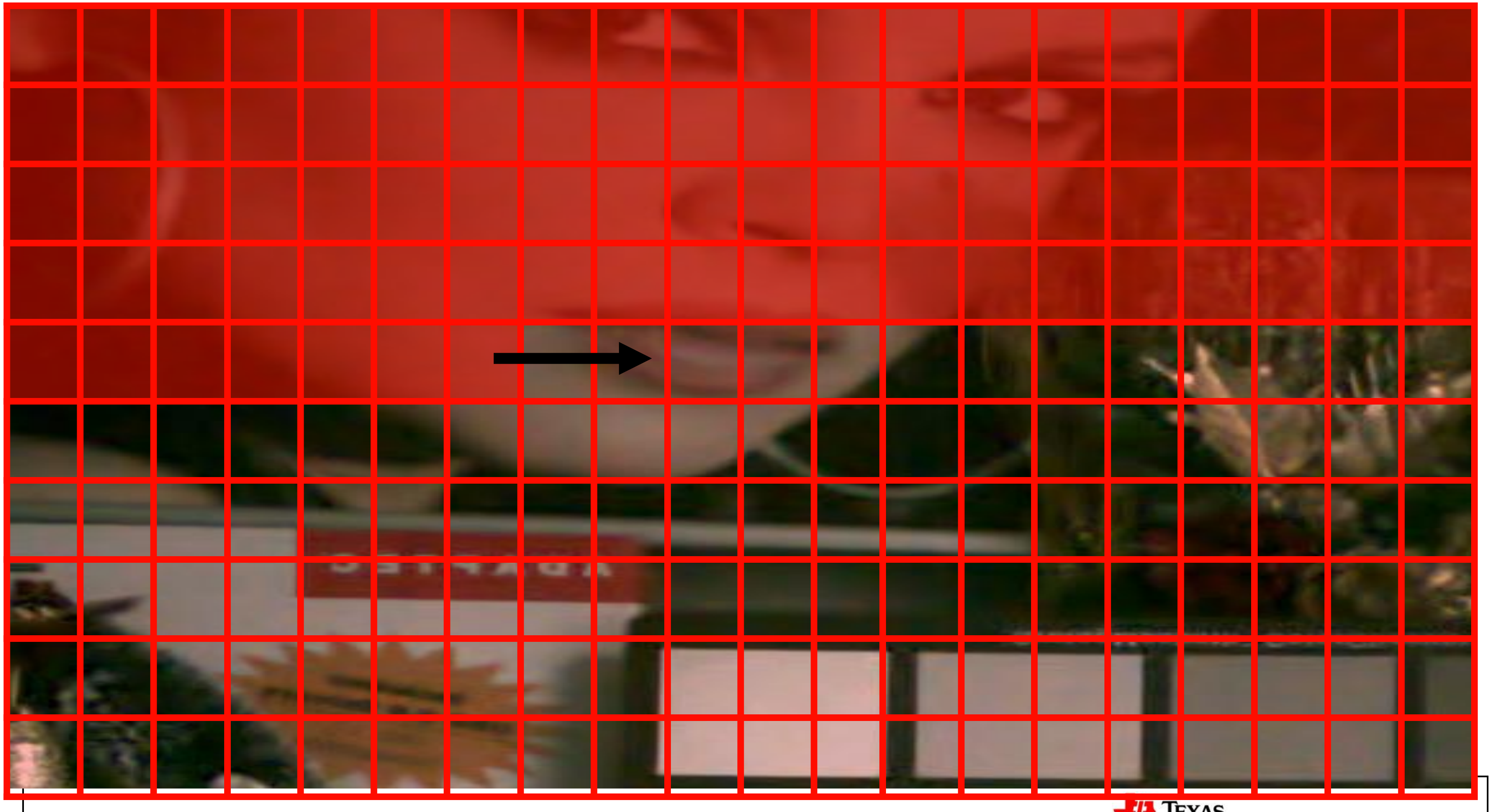
-ARP32 executes a VLOOP to process data in the ping buffer (ex: image buffer B)

Ping-Pong buffering between image buffer A & B allows VCOP to process one set of data while the other set is being transferred in or out.

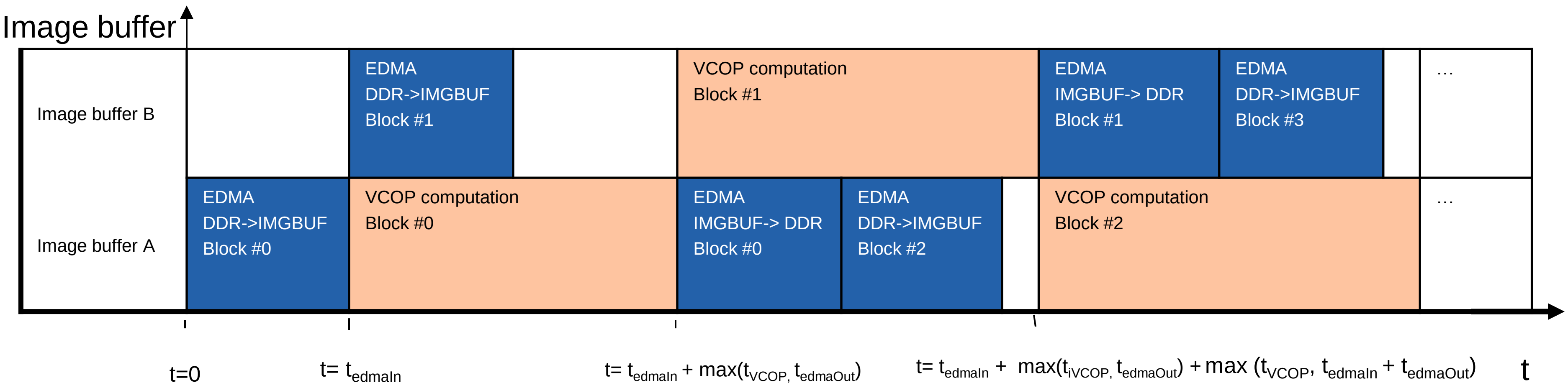
ImBuf A & B serve as I/O buffer whereas the working buffer is always connected to VCOP.

Block based processing

Due to size of image buffer, VCOP can only operate on 32 kb of data at a time. The original image is divided in blocks. To process the entire image by VCOP, every block of the image is transferred from DDR to image buffer, processed by the VCOP and then transferred back to DDR.



Parallelizing memory transfer and VCOP computation



Concurrent processing Graph

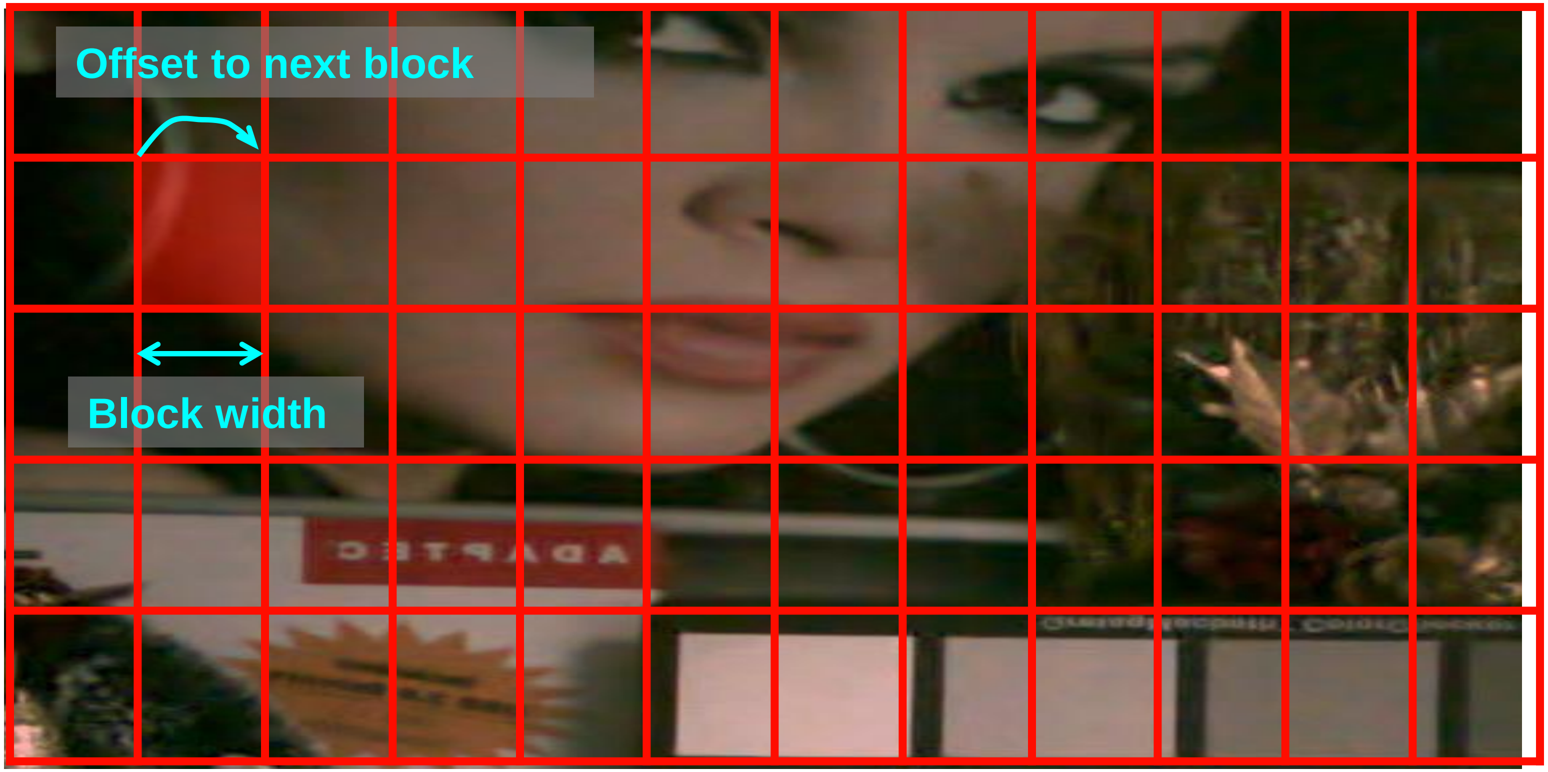
Total execution time for one block is $t = \max(t_{VCOP}, t_{edmaIn} + t_{edmaOut})$

Block based processing

Block size can be various, not necessarily square. Only restriction is that they must fit within the 32kb of the image buffer.

Block based processing

Offset to next block and block width are two independent parameters.
Example: **Offset to next block = block width**



Block based processing

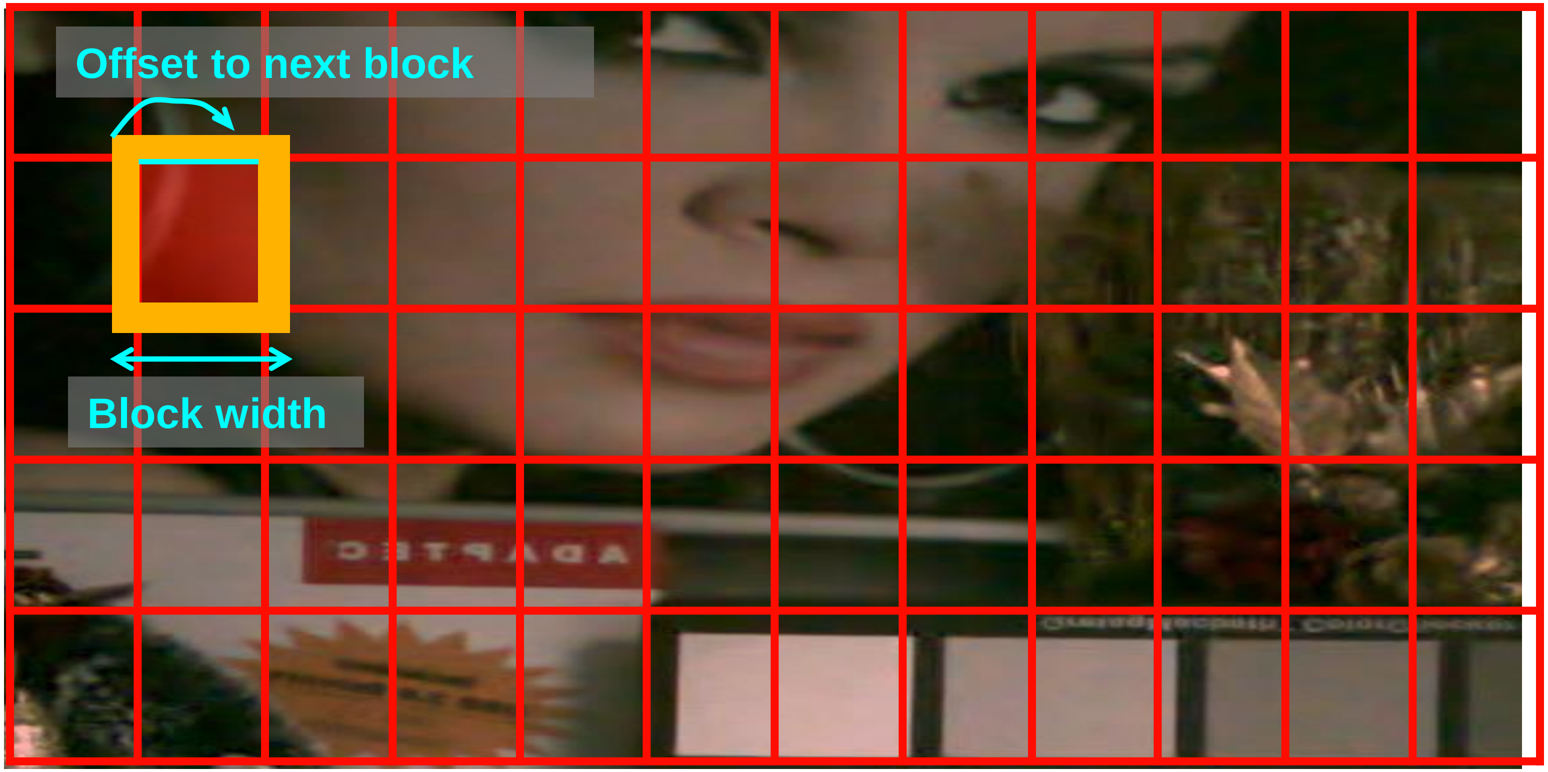
Offset to next block and block width are two independent parameters.
Example: **Offset to next block = block width**

Block based processing

Offset to next block is not necessarily equal to block width in order to transfer overlapping blocks. Overlapping blocks are used in case of filtering to account for the overlapping border pixels. For instance a N taps filter needs N-1 border pixels. The border width would be (N-1)/2 pixels.
Example:

Offset to next block= w

block_width= w + N-1

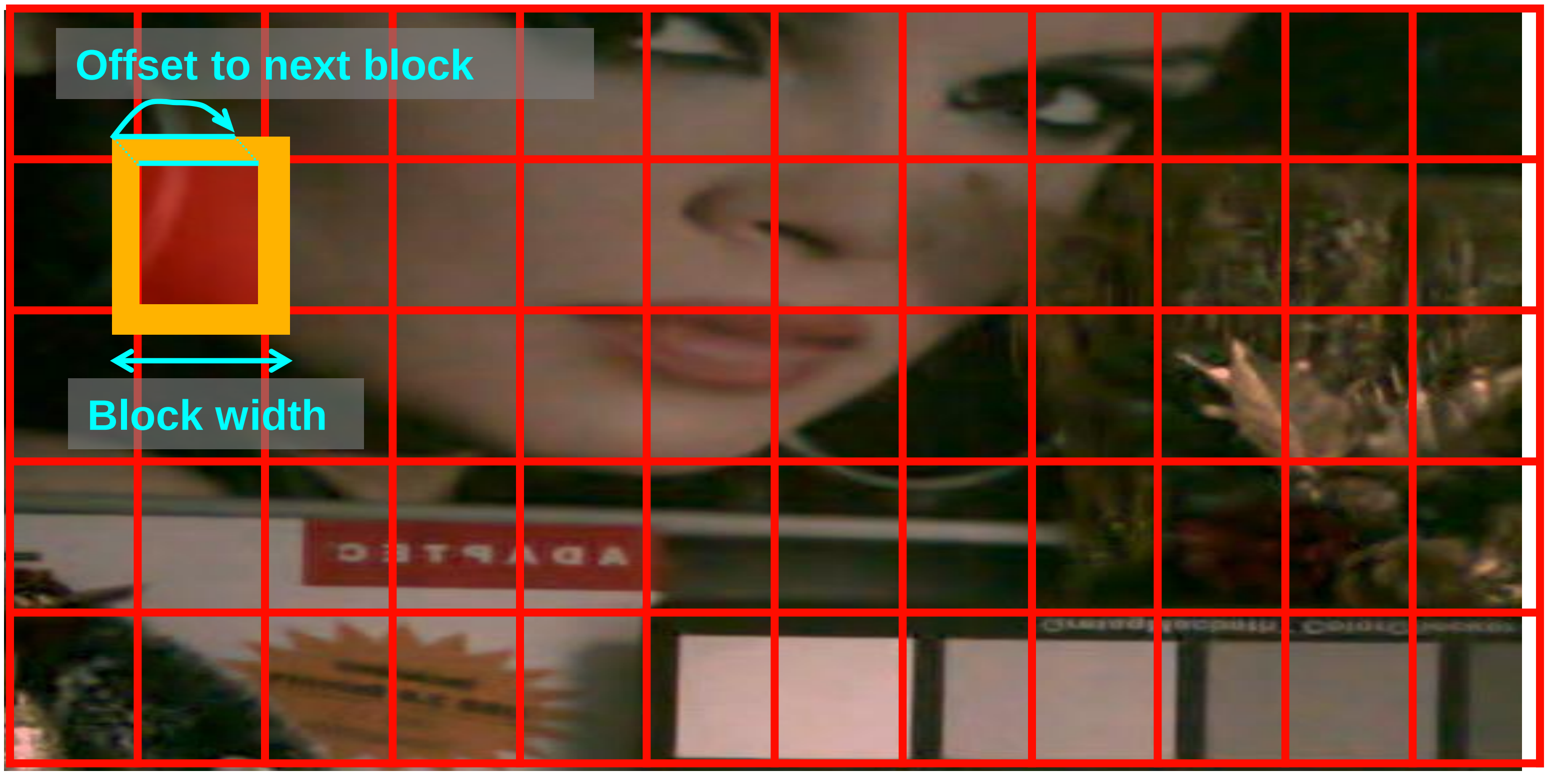


Block based processing

Offset to next block is not necessarily equal to block width in order to transfer overlapping blocks. Overlapping blocks are used in case of filtering to account for the overlapping border pixels. For instance a N taps filter needs N-1 border pixels. The border width would be (N-1)/2 pixels.
Example:

Offset to next block= w

block_width= w + N-1



Block based processing

Offset to next block is not necessarily equal to block width in order to transfer overlapping blocks. Overlapping blocks are used in case of filtering to account for the overlapping border pixels. For instance a N taps filter needs N-1 border pixels. The border width would be (N-1)/2 pixels.
Example:

Offset to next block= w

block_width= w + N-1

Scheduling code

In *eve_algo_dma_auto_incr.c*, 2 functions:

- `EVELIB_algoDMAAutoIncrSequential()`: implement block-based sequential processing for debugging.

- `EVELIB_algoDMAAutoIncrConcurrent()`: implement block-based parallel processing for production code.

These functions follow a pre-programmed DMA access patterns that was defined during a setup phase. Because we have a setup phase, execution of the scheduling is done very quickly by the sequence of calls:

```
VCOP_BUF_SWITCH_TOGGLE()
EDMA_UTILS_autoIncrement_triggerOutChannel()
execFunc[k]
EDMA_UTILS_autoIncrement_waitOutChannel()
EDMA_UTILS_autoIncrement_waitInChannel()
```

DMA Setup code

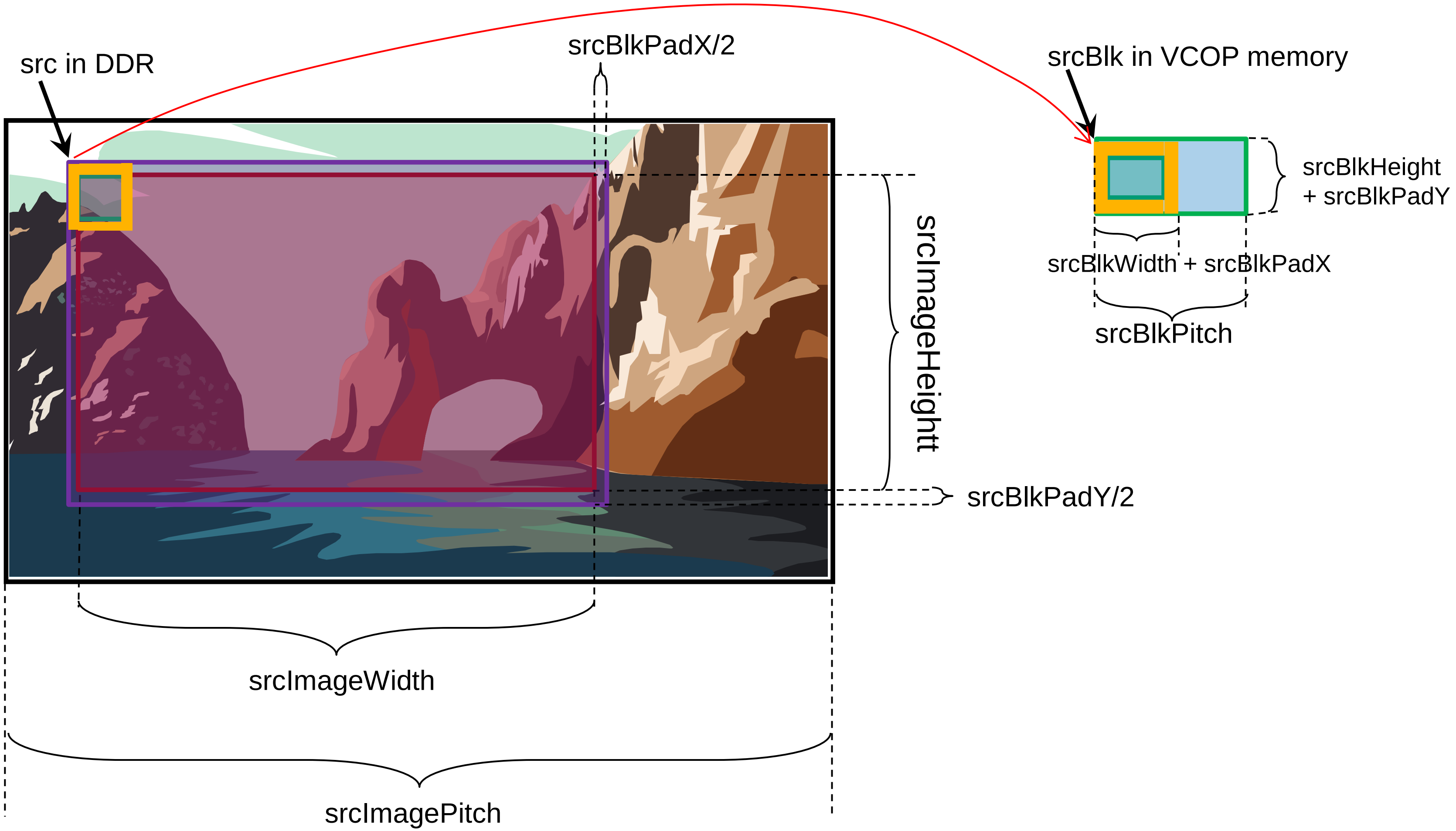
Setup code for DMA is done in *evelib_fir_filter_2d.c* by calling `EVELIB_algoDMAAutoIncrInit()` implemented in *eve_algo_dma_auto_incr.c.*

```
int EVELIB_algoDMAAutoIncrInit(
    unsigned char       *src,
    unsigned int        srcImageWidth,
    unsigned int        srcImageHeight,
    int                 srcImagePitch,
    unsigned char       *dst,
    unsigned int        dstImageWidth,
    unsigned int        dstImageHeight,
    int                 dstImagePitch,
    unsigned char       *srcBlk,
    unsigned int        srcBlkWidth,
    unsigned int        srcBlkHeight,
    int                 srcBlkPitch,
    unsigned char       *dstBlk,
    unsigned int        dstBlkWidth,
    unsigned int        dstBlkHeight,
    int                 dstBlkPitch,
    unsigned int        srcBlkPadX,
    unsigned int        srcBlkPadY)
```

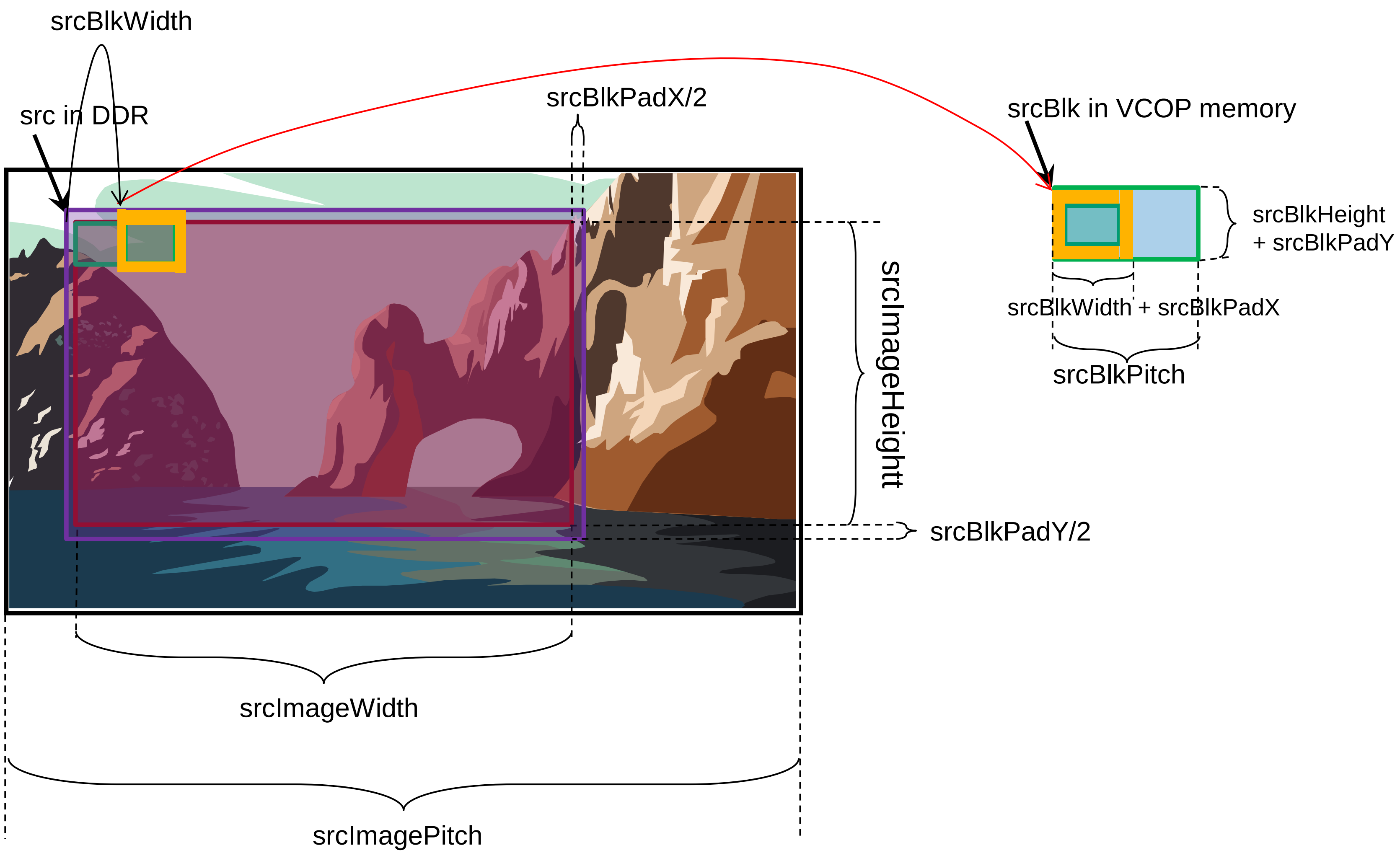
In external memory (DDR) — src through dstImagePitch

In VCOP memory — srcBlk through dstBlkPitch

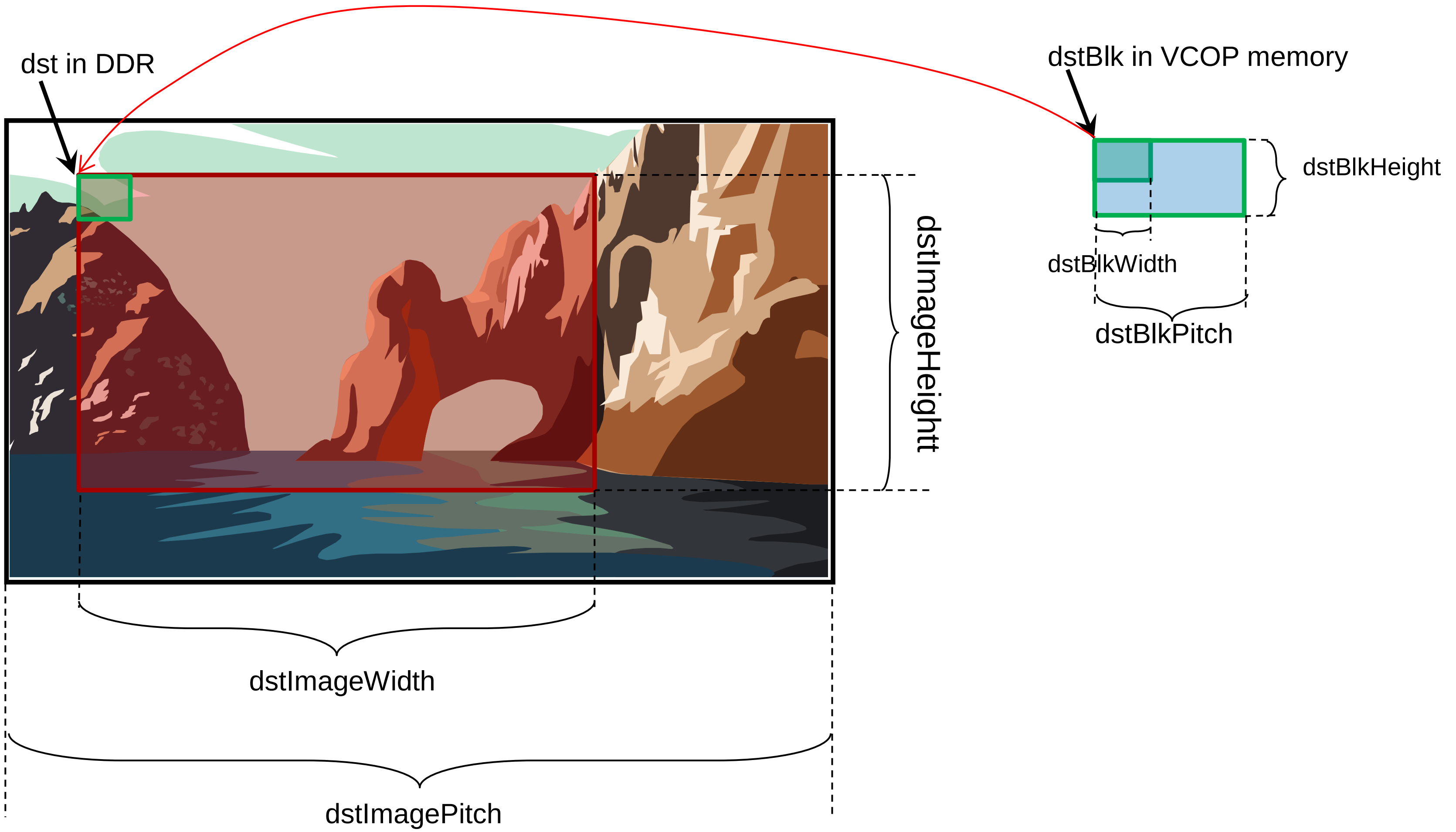
DMA Setup code



DMA Setup code



DMA Setup code



DMA Setup code

The underlying Starterware functions are `EDMA_UTILS_autoIncrement_init()`, `EDMA_UTILS_autoIncrement_configure()`, which are used to implement `EVELIB_algoDMAAutoIncrInit()`:

```
    initParam.numInTransfers    = 1;
   initParam.numOutTransfers   = 1;
   initParam.transferType      = EDMA_UTILS_TRANSFER_INOUT;

   initParam.transferProp[0].roiWidth             = srcImageWidth+srcBlkPadX;
   initParam.transferProp[0].roiHeight            = srcImageHeight+srcBlkPadY;
   initParam.transferProp[0].roiOffset            = 0;
   initParam.transferProp[0].blkWidth             = srcBlkWidth+srcBlkPadX;
   initParam.transferProp[0].blkHeight            = srcBlkHeight+srcBlkPadY;
   initParam.transferProp[0].extBlkIncrementX     = srcBlkWidth;
   initParam.transferProp[0].extBlkIncrementY     = srcBlkHeight;
   initParam.transferProp[0].intBlkIncrementX     = 0;
   initParam.transferProp[0].intBlkIncrementY     = 0;
   initParam.transferProp[0].extMemPtrStride      = srcImagePitch;
   initParam.transferProp[0].interMemPtrStride    = srcBlkPitch;
   initParam.transferProp[0].extMemPtr            = src;
   initParam.transferProp[0].interMemPtr          = srcBlk;
   initParam.transferProp[0].dmaQueNo             = 0;

   initParam.transferProp[1].roiWidth             = dstImageWidth;
   initParam.transferProp[1].roiHeight            = dstImageHeight;
   initParam.transferProp[1].roiOffset            = 0;
   initParam.transferProp[1].blkWidth             = dstBlkWidth;
   initParam.transferProp[1].blkHeight            = dstBlkHeight;
   initParam.transferProp[1].extBlkIncrementX     = dstBlkWidth;
   initParam.transferProp[1].extBlkIncrementY     = dstBlkHeight;
   initParam.transferProp[1].intBlkIncrementX     = 0;
   initParam.transferProp[1].intBlkIncrementY     = 0;
   initParam.transferProp[1].extMemPtrStride      = dstImagePitch;
   initParam.transferProp[1].interMemPtrStride    = dstBlkPitch;
   initParam.transferProp[1].extMemPtr            = dst;
   initParam.transferProp[1].interMemPtr          = dstBlk;
   initParam.transferProp[1].dmaQueNo             = 0;

   status = EDMA_UTILS_autoIncrement_init(autoIncrementContext,&initParam);
```

EVE Setup code

In evelib_fir_filter_2d.c, EVE kernels are initialized and memories are allocated:

```
EVELIB_KernelFuncType execFunc[] =
{(EVELIB_KernelFuncType)vcop_filter_uchar_char_uchar_vloops};
    EVELIB_KernelContextType context[] =
{(EVELIB_KernelContextType)__pblock_vcop_filter_uchar_char_uchar};
    unsigned int numKernels = 1;

VCOP_BUF_SWITCH_SET (WBUF_SYST, IBUFHB_SYST, IBUFLB_SYST, IBUFHA_SYST, IBUFLA_SYST);

srcBlk = (unsigned char *)vcop_malloc(VCOP_IBUFLA, srcBlkPitch * srcBlkHeightTot);
dstBlk = (unsigned char *)vcop_malloc(VCOP_IBUFHA, dstBlkPitch * dstBlkHeight);
coeffBlk = (char *)vcop_malloc(VCOP_WMEM, coeffH * coeffW);

memcpy(coeffBlk, coeff, coeffH * coeffW);

vcop_filter_uchar_char_uchar_init(srcBlk, coeffBlk, dstBlk, srcBlkPitch, coeffW,
coeffH, srcBlkWidth, srcBlkHeight, dnsmplVert, dnsmplHorz, rndShift,
__pblock_vcop_filter_uchar_char_uchar);
```