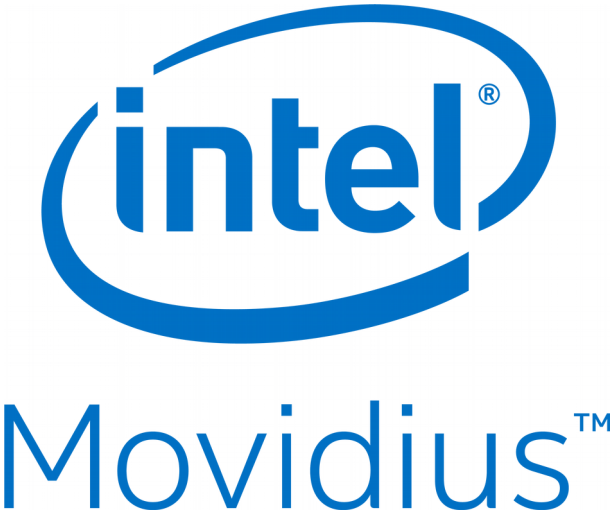


MA2x5x SIPP User Manual

v1.32 / July 2018

Intel® Movidius™ Confidential

Copyright and Proprietary Information Notice

Intel Movidius
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Revision History

Date	Version	Description
July 2018	1.32	Minor linguistic and formatting corrections. Updated logo.

Table of Contents

1 General Overview

1.1 About this document

This document describes the Movidius SIPP (Streaming Image Processing Pipeline) MA2x5x framework. It is created as a replacement for the existing MA2100 SIPP User Guide which introduced the SIPP MA2100 framework. Naturally there is a lot of commonality between the inherent functionality and motivations of both frameworks. Where appropriate information from MA2100 SIPP User Guide has been republished within this document so that it works as a standalone publication and a direct replacement for MA2100 SIPP User Guide.

Differences in the SIPP MA2x5x framework versus its predecessor will be illustrated within. Further provided is the layout of a feature plan for the future development of the framework and to a comprehensive guide to the usage of the framework on MA2x5x silicon.

1.2 Related documents and resources

Related documentation can be obtained from http://www.movidius.org. If you do not have access to the documents below, you can request them. Relevant documents include:

1. Myriad 2 Development Kit (MDK) – Getting Started Guide.
2. Myriad 2 Development Kit (MDK) – Programmer's Guide.
3. Myriad 2 Development Kit (MDK) – MA2100 SIPP User Guide.
4. MoviEclipse (MDK – Tools documentation).

1.3 Notational conventions

The following is a description of some of the notations used in this document.

1.3.1 Data formats

Format	Description
U8	Unsigned 8-bit integer data
U16	Unsigned 16-bit integer data
U32	Unsigned 32-bit integer data
I8	Signed 8-bit integer data
I16	Signed 16-bit integer data
I32	Signed 32-bit integer data
10P32	10-bit RGB packed into 32 bits (xxRRRRRRRRRGGGGGGGGGGBBBBBBBBBB)
FP16	IEEE-754 16-bit floating point (half precision, 16-bit)

Format	Description
FP32	IEEE-754 32-bit floating point (single precision, 32-bit)
U8F	Unsigned 8 bit fractional data the range [0, 1.0]

Table 1: SIPP Data Formats

1.3.2 Fixed point formats

Fixed-point data may be either signed, or unsigned. It has one or more bits of fractional data, and 0 or more bits of integer. For signed data, the specified number of integer bits includes the signed bit.

Examples:

- U8.8: Unsigned, with 16 bits of storage (8 bits of integer and 8 fractions).
- S16.16: 1 sign bit, 15 bits of integer precision, and 16 bits of fractional data.

The data can be interpreted by treating it as integer, then dividing by 2^N, where N is the number of fractional bits.

1.3.3 Glossary terms

When a term that is defined in the glossary is used for the first time in the document, it will be written in *italics*.

Term	Description
AWB	Auto While Balance
Bayer	A particular CFA layout, whereby the color channels are arranged in the image as a matrix of 2x2 blocks. Within each block there are two diagonally-opposed green pixels, as well as a red and a blue pixel. The image can be thought of as a 4-channel image, with the channels labeled Gr, R, B and Gb. Green pixels on lines where there are red pixels belong to the Gr channel, whereas green pixels on lines where there are blue pixels belong to the Gb channel.
Bayer Order	Describes the layout of a 2x2 block of pixels in a Bayer image. Depending on which color channel is located at the top-left of the image, the Bayer Order will be one of GRBG, GBRG, RGGB or BGGR.
CFA	Color Filter Array
CMX	Low-latency, high bandwidth memory and cross-connect subsystem
CSI	Camera Serial Interface – a physical serial interface defined by the MIPI Alliance for connecting camera devices to Application Processors.
DAG	Directed Acyclic Graph
Filter	A SIPP filter is an entity which does pixel-level processing within a SIPP pipeline. Filters may be instantiated as nodes in a SIPP pipeline graph. The same type of filter may be

Term	Description
	instantiated more than once in a graph.
MIPI	**M**obile **I**ndustry **P**rocessor **I**nterface. The MIPI Alliance is a standards organization focused on specifying interfaces between hardware components on mobile devices, such as CSI.
Output Buffer	An output buffer is a circular line buffer, used to store the processed data output by a filter.
Inline processing	When an application performs all processing without buffering any data in DDR, it is called Inline Processing. An example of inline processing would be a system which processes lines of data as they arrive from a camera sensor, where all line buffering is in local CMX memory. As processed lines of data become available, they are transmitted directly from a local CMX memory buffer to the output device, by a sink filter.
ISP	Image signal processing. Typically refers to the processing of image streams coming from digital camera sensors.
SHAVE	**S**treaming **H**ybrid **A**rchitecture **V**ector **E**ngine. Vector processing cores used in Movidius Myriad processors.
Sink Filter	A filter in a SIPP pipeline which does not have any children. These filters typically output data to an external entity, such as DRAM (using a DMA controller) or a display controller.
SIPP	Streaming Image Processing Pipeline
Source Filter	A filter in a SIPP pipeline which does not have any parents. These filters typically source data from an external entity, such as DRAM (using a DMA controller) or a camera interface.

Table 2: Glossary

2 Introduction to the SIPP framework

2.1 Motivation

Many image processing libraries, such as OpenCV, perform a series of whole-frame operations in series. This is very DDR intensive, since an entire set of frames must be read from and/or written back to DDR for each operation. Performance is typically limited by available DDR bandwidth. This is mitigated on x86 platforms by the presence of large CPU caches, but for mobile systems with low-power requirements, it is not a suitable paradigm.

The approach used by SIPP involves a graph of connected filters. Data is streamed from one filter to the next, on a scanline-by-scanline basis. Images are consumed in raster order. Scanline buffers are located in low-latency local memory (CMX). No DDR accesses should be necessary (other than accessing any pipeline input or output images located in DDR, using DMA copies to/from local memory). In addition to the performance and power benefits of avoiding DDR accesses, the design can also reduce hardware costs, allowing stacked DDR to be omitted for certain types of applications.

2.2 Myriad SoCs

The SIPP framework is designed to maximize the usage of the available processing resources in Myriad SOCs. On MA2150 silicon, there are 12 SHAVE vector processing cores. Also available are DMA controllers, for transferring data from DDR to CMX, and vice-versa. Additionally, a number of hardware accelerators, for some computationally expensive ISP and computer vision tasks are provided.

Whereas the bulk of the processing is performed by the SHAVE cores and the hardware accelerators, the SIPP framework runs on a RISC processor.

The SIPP environment is also the development framework for the MA2150 Media sub-system which is a collection of SIPP accelerators consisting of a complementary collection of hardware image processing filters designed primarily for use within the SIPP software framework, allowing generic or extremely computationally intensive functionality to be offloaded from the SHAVEs.

CMX memory is generally also used to implement input and output buffers for the hardware filters. An arbitrary ISP pipeline may then be flexibly defined in software but constructed from both software resources – filter kernels implemented on the SHAVEs – and high performance dedicated hardware resources. IN the majority of occasions, CMX memory provides the means of connecting up consecutive stages of a pipeline: one filter's output buffer is another's input buffer.

2.3 Filter graphs

Processing under the SIPP framework is performed by *filters*. Applications construct pipelines consisting of filter nodes linked together in a *DAG* (Directed Acyclic Graph). Each filter is coupled with one or more *output buffers*. The output buffer stores the processed data output by the filter, and can store zero or more lines of data (zero lines in the case of a *sink filter*). When a filter is invoked, it produces at least one new line of data in its output buffer. Certain runtimes enable more than one new line to be produced per invocation

of the filter. Lines are added to the output buffer in a circular fashion: the lines are written at increasing addresses, until the end of the buffer is reached, at which point the output position wraps back to the start of the buffer.

2.3.1 Filter graph rules

The graph validity rules are as follows:

- A filter is allowed to have multiple parent nodes. That is, a filter may source data from more than one buffer.
- A filter is allowed to have multiple child nodes. That is, more than one filter may source data from a filter's output buffer.
- A filter that has no parents (a *source filter*) must have at least one child node.
- A filter that has no children (a *sink filter*) must have at least one parent node.
- A source filter connected directly to a sink filter, with no filter(s) in between, is not permitted.
- The graph need not be connected. An example of such a pipeline would be one where incoming frames consist of separate Luma and Chroma planes, and where the Luma and Chroma processing paths are completely independent (see Figure 3: An example of a disconnected graph).
- For R1 Release of the framework, no hardware filter may appear twice in any graph (See section 6.1 – deltas in MA2x5x SIPP FW versus MA2100 SIPP FW)

The application may construct the graph programmatically by making SIPP API calls. The application performs the following steps:

1. Instantiates the pipeline.
2. Instantiates the required filters within the pipeline.
3. Connects the filters together to form the graph, by specifying parent/child relationships.

The pipeline is now ready to be executed. When the application initiates execution of the pipeline for the first time, the framework will first automatically allocate memory for the output buffers. The size of an output buffer is calculated based on the requirements of the filters consuming from that buffer (for example, a filter applying a 7x7 convolution requires 7 at least 7 lines to be present in the output buffer). Additionally, if the graph has multiple paths which diverge and subsequently converge, extra buffering may be required on one of the paths to ensure that the data is synchronized when it reaches the convergence point. The framework automatically determines what extra lines of buffering are needed, and allocates the buffer memory accordingly.

2.3.2 Simple pipeline examples

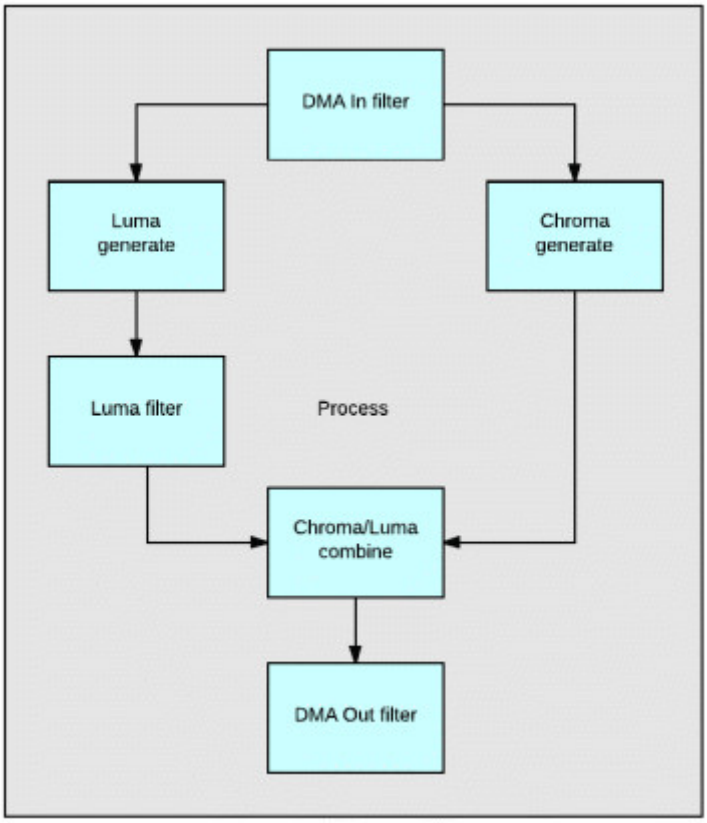


Figure 1: Simple pipeline

In the above example, since the Luma processing path is longer than the Chroma processing path, the framework automatically adds extra lines to the “Chroma generate” filter’s output buffer, so that the Luma and Chroma data is in sync when it arrives at the “Chroma/Luma combine” filter. Adding extra buffering lines allows the latency of the alternate paths to be matched.

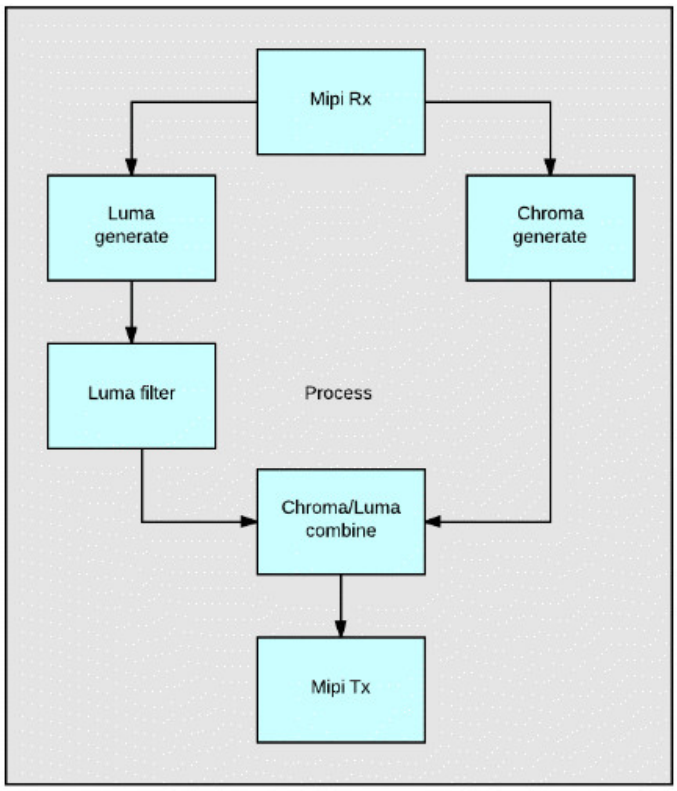


Figure 2: Simple pipeline without DDR, streaming from MIPI sensor

Both of the previous examples perform the same data processing. However, the above example does not require any DDR. The data can be processed in a streaming fashion, using only local memory. Data coming from a camera is stored in a local memory buffer by the Mipi Rx filter (in the Mipi Rx filter's output buffer). The processed data is then transmitted directly from the Chroma/Luma combine filter's output buffer by the Mipi Tx filter. This mode of operation, which doesn't require DDR, is known as *inline processing***.** An application which does all of its processing inline may be run on a Myriad processor that has not been packaged with stacked DDR.

Note that the above pipeline can operate in a fully "streaming" fashion: that is, data can be processed in-line as it arrives from the sensor, adding minimal latency to the data path. All buffers are located in local (CMX) memory, meaning that this pipeline can run on a processor that is not packaged with external DDR.

2.3.3 Superpipes

A superpipe is a pipeline which consists of multiple disconnected pipelines. While the datapaths for each pipeline within the superpipe are completely separate as far as the SIPP framework is concerned, there is only a single pipeline, and only a single schedule needs to be computed. Building a superpipe is no different from building any other type of pipeline – since there is no requirement that the graph be connected, a superpipe is a valid form of SIPP pipeline.

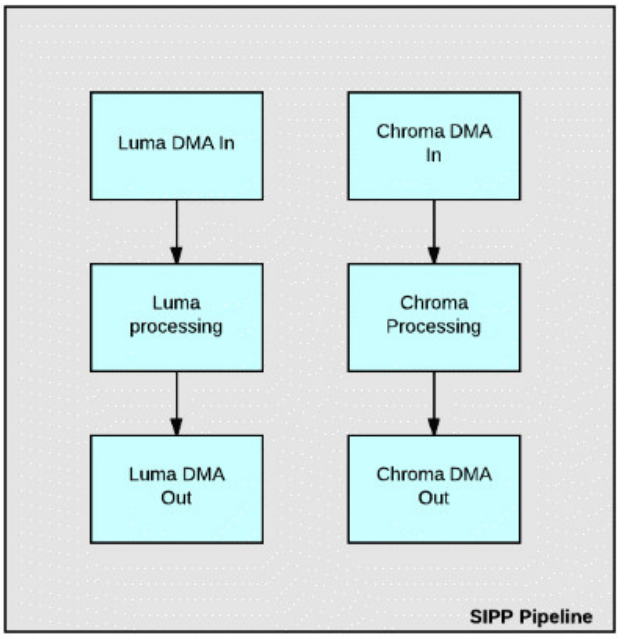


Figure 3: Example of a Superpipe – Disconnected Graph

2.3.4 Data rate matching

For most filters, the size of the output image matches the size of the input image. Therefore, the number of lines that arrive in a filter's input buffers (its parent's output buffers) during the course of processing a frame is normally equal to the number of lines it produces it its output buffer. However, this is not true for certain filters, such as filters which perform a resizing operation. Consider for example, a filter which down-samples an image by a factor of two in both the horizontal and vertical directions. In the horizontal direction, the width of each line in the filters output buffer will be half the width of the lines in the parent's output buffer. In the vertical direction however, what happens is that the downsizing filter only runs once, producing a single line of data in its output buffer, for every two lines produced in its parent's output buffer. The SIPP framework manages this scheduling automatically, making sure that the resize filter is not invoked until the correct lines are present in the parent's output buffer.

This means that the line rate may not be the same for all parts of the pipeline. Particular care has to be taken when different paths in the pipeline converge. The line arrival rate at the inputs to the filter at the convergence point must match what that filter expects. This is best illustrated by way of an example.

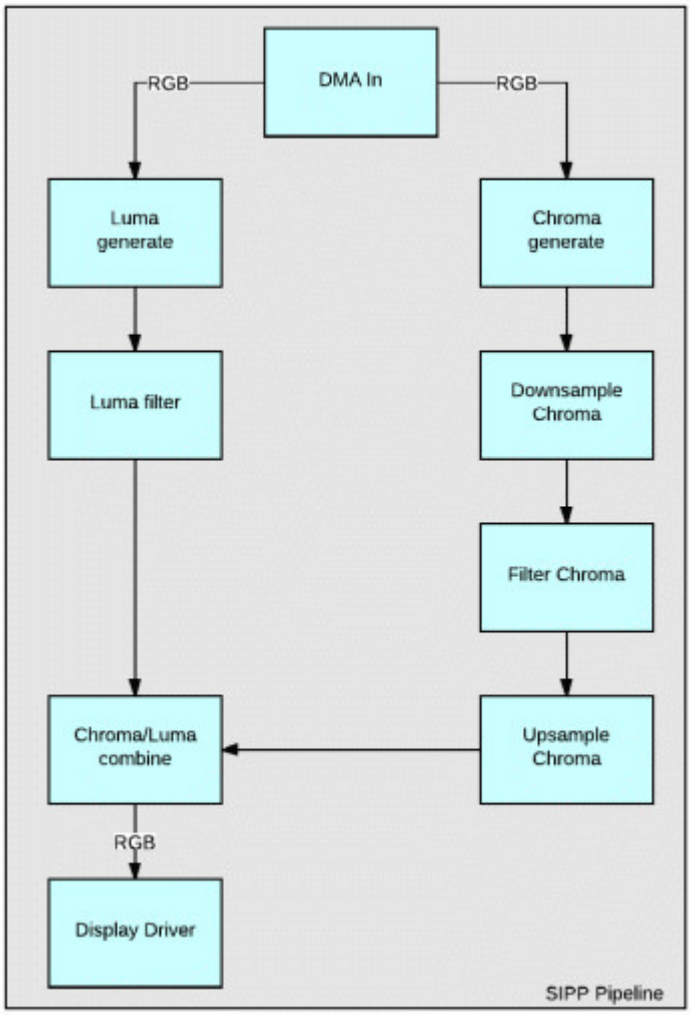


Figure 4: Filters with different output line rates

In the example above, the “Downscale Chroma” filter downsamples the data by a factor of 2 in each dimension. The “Filter Chroma” filter operates on this subsampled data, without altering the image size. The “Downscale Chroma” and “Filter Chroma” filters only get scheduled half as often as the other filters: in the course of processing a frame, they only produce half the number of lines. The “Downscale Chroma” filter only runs once for every two lines that are added into its parent’s output buffer. The “Upsample Chroma” filter, on the other hand, runs twice for every line that is output into its parent’s output buffer. The net effect is that the line output rate of the “Upsample Chroma” filter matches the line output rate of the Luma filter, allowing the “Chroma/Luma combine” filter to merge the data coming from the two paths.

Note also that it would be possible to merge the Chroma Upsampling filter into the “Chroma/Luma combine” filter, in order to save memory and local memory bandwidth. This is possible as long as the combine filter consumes the data from its Chroma input at half the rate that it consumes data from its Luma input.

2.3.5 Hardware and software filters

Some filters are drivers for hardware interfaces. For example, a filter could drive a DMA controller, or a display or camera interface, in order to source or output data. Additionally on Myriad 2, a filter could be a driver for a Hardware Accelerator. The remaining types of filters are Software Filters. Software filters are

implemented on the Shave processors. They can be implemented in C, assembly language, or a mixture of both.

A pipeline can be implemented as a mixture of hardware and software filters. Multiple instances of the same software filter can be used in a pipeline. HW filters may be used only once.

2.3.6 Multiple pipelines and DDR

For more complex applications, it's possible to instantiate multiple pipelines. For example, you might want to run an ISP (Image Signal Processing) pipeline on the image coming from the camera, and run a Computer Vision application on the resulting image stream. Or, alternatively, there might be two camera inputs in the system, with a SIPP pipeline instantiated to process data coming from each of the cameras.

We cannot of course construct arbitrarily complex pipelines, or run an arbitrary number of pipelines concurrently because we will at some point run out of local memory. We can solve this problem if stacked DDR is available. If we have a single, very complex pipeline, and we run out of local memory, we can split the pipeline into two or more pipelines. To get around the local memory limit, we do not run our multiple pipelines in parallel. Instead, we process an entire frame with the first pipeline, with the output frame(s) being written to DDR. Then we process an entire frame with the second pipeline. This scheme works because the contents of the line buffer memory does not need to be preserved from one frame to the next. The amount of extra DDR traffic generated is small: the filters that do all of the real work are still operating from local memory. If the input data is coming into the application from a real-time source, such as a camera, we might also need to add some DDR buffering at input, since the pipeline which is directly consuming the data from the camera is not running all of the time.

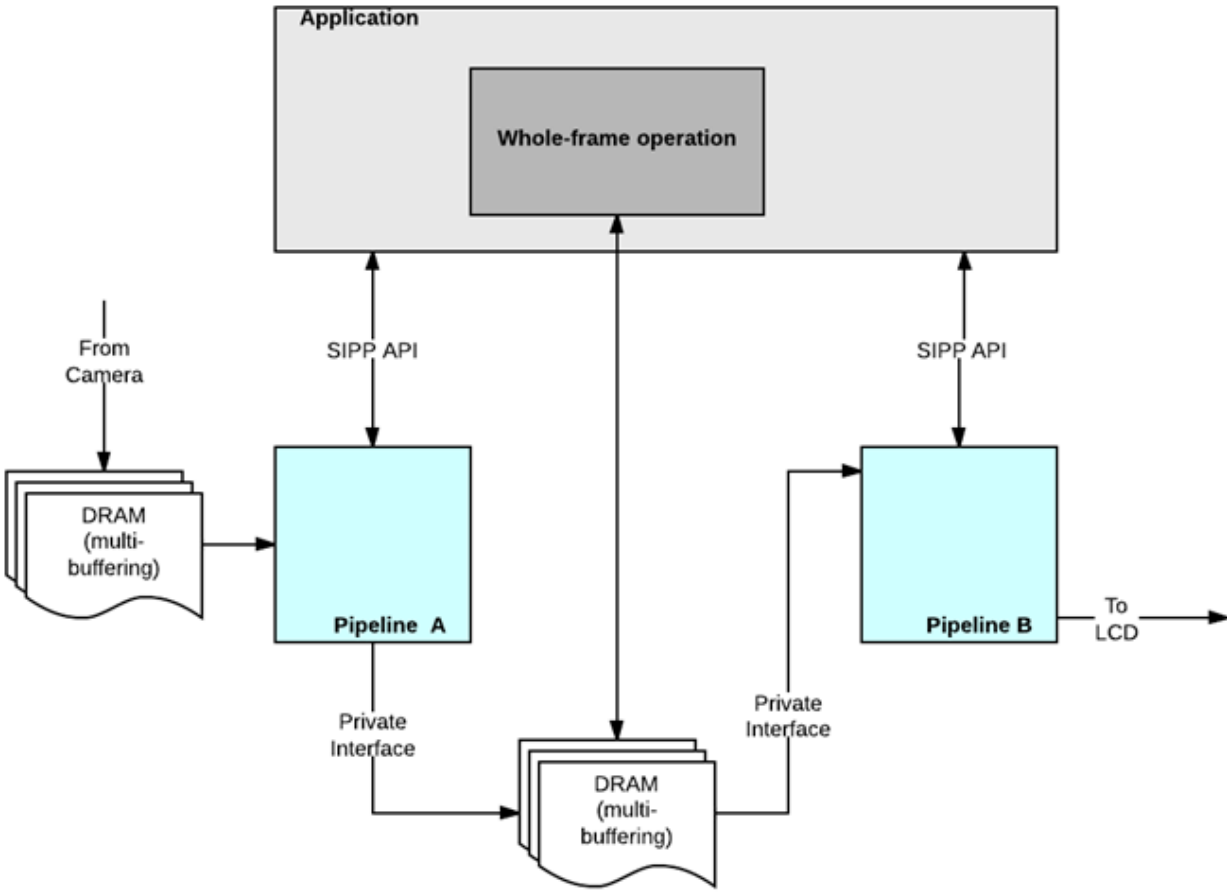


Figure 5: An application with multiple SIPP pipelines

Another reason for splitting the pipeline is if part of the application doesn't lend itself to raster-based processing. An example would be if some whole-frame operation were to be performed, such as a 2D FFT, or a rotation by an arbitrary angle. The whole-frame operation could be performed externally to the SIPP framework, as in the diagram above.

2.4 General framework architecture

Figure 6 Illustrates a basic overview of the SIPP architecture in terms of the main functional aspects involved in launching a pipeline onto the underlying HW and Shave resource.

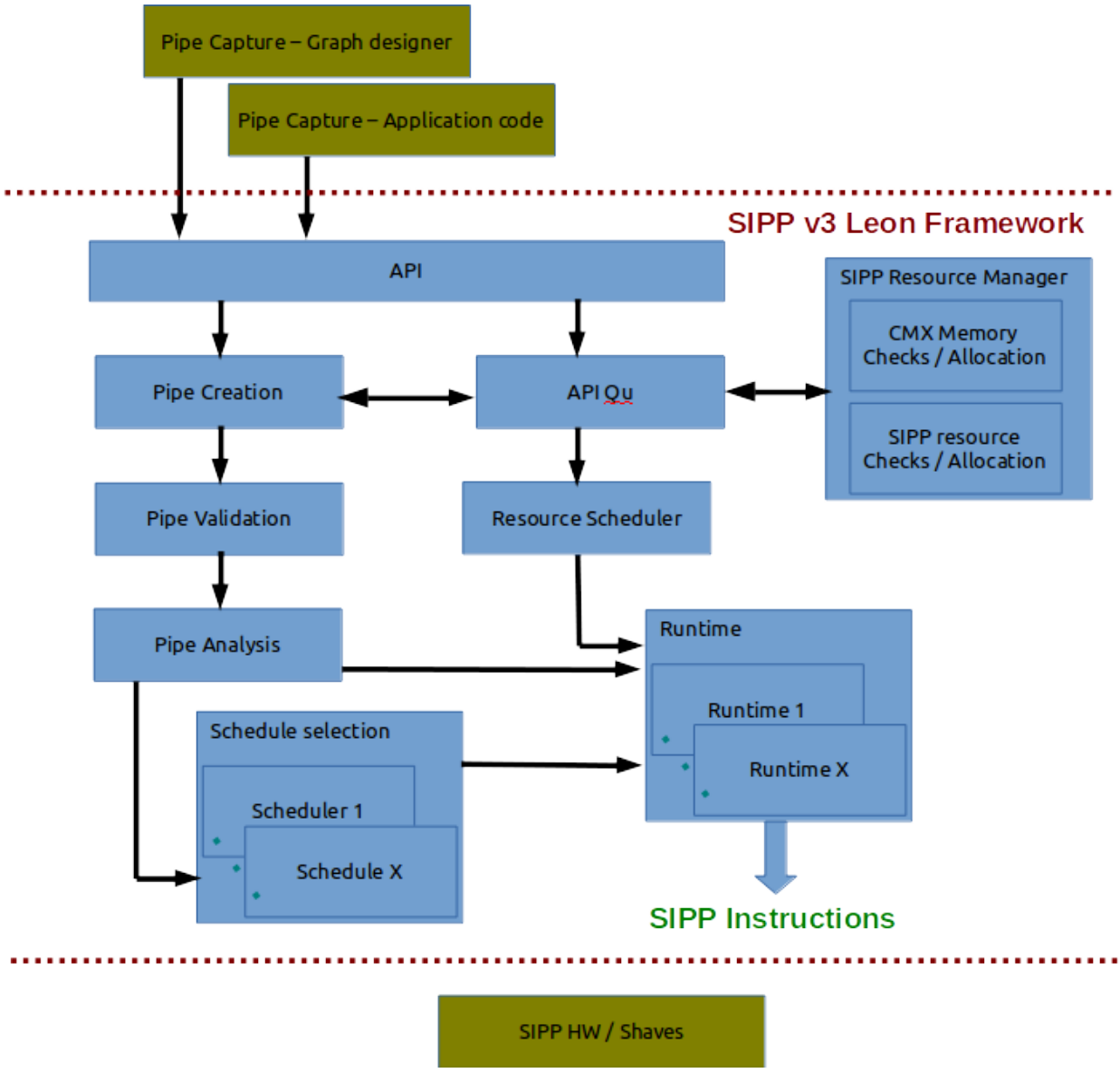


Figure 6: General SIPP framework architecture

- API: The API as presented in chapter 7.
- Pipe Creation: Allocates the storage for the pipeline and filter description structures and any associated parameters and implements the connections between filters as guided by the API.
- Pipe Validation: After creation of a pipe, the pipeline will be checked against a set of rules to ensure it is considered valid.
- Pipe Analysis: The main function of this unit is to establish the most suitable scheduler and

runtime to be used for the pipeline. Different pipelines have different features and to enable a framework which is both flexible and efficient a suite of schedulers and runtimes are available. This unit will establish which is considered most suitable for a pipeline and will on the basis of this result perform some other concomitant tasks to enable efficient operation within the selections.

- API Queue: An internal queue stacking most recent operations for each pipeline. Since some APIs use a non-blocking approach, they are held in this queue until selected by the resource scheduler for execution.
- Resource Scheduler: Manages access to the underlying SIPP HW and Shave resource – This will try to ensure all resource is kept as busy as possible based on the pending operations and their ability to run together at the same time. For those operations which cannot be executed together (due to an underlying resource conflict) this unit will manage appropriate scheduling for each pipeline.
- SIPP Resource Manager: Ensures each pipeline has access to the memory and other resources it requires and is responsible for collecting that resource again on termination of the pipeline.
- Scheduler: Creates a schedule (when required) for a pipeline. Each iteration of a schedule contains instructions on which HW and SW filters are required to operate on that iteration. In effect this schedule acts as the input to the runtime when it is executing the pipeline on the HW /Shave resource. The schedule creation process ensures that a minimum amount of line buffer data needs to be stored in the allocated CMX resource while ensuring no data dependency between filters involved within an iteration. Depending on the features of the pipeline, schedule creation may be complex. It is expected that the Pipe analysis stage will pick the schedule most suited to the pipeline to perform the scheduling most efficiently.
- RunTime: Taking the schedule as an input the runtime manages the execution of the filter kernels on the HW and SHAVE units, iteration by iteration until the full operation is complete.

3 Using the SIPP framework

3.1 Building a SIPP application

The simplest way to create SIPP pipeline application is to use the Graph Designer [4]. This is a plug-in to the moviEclipse IDE that allows users create, build and execute applications from the IDE.

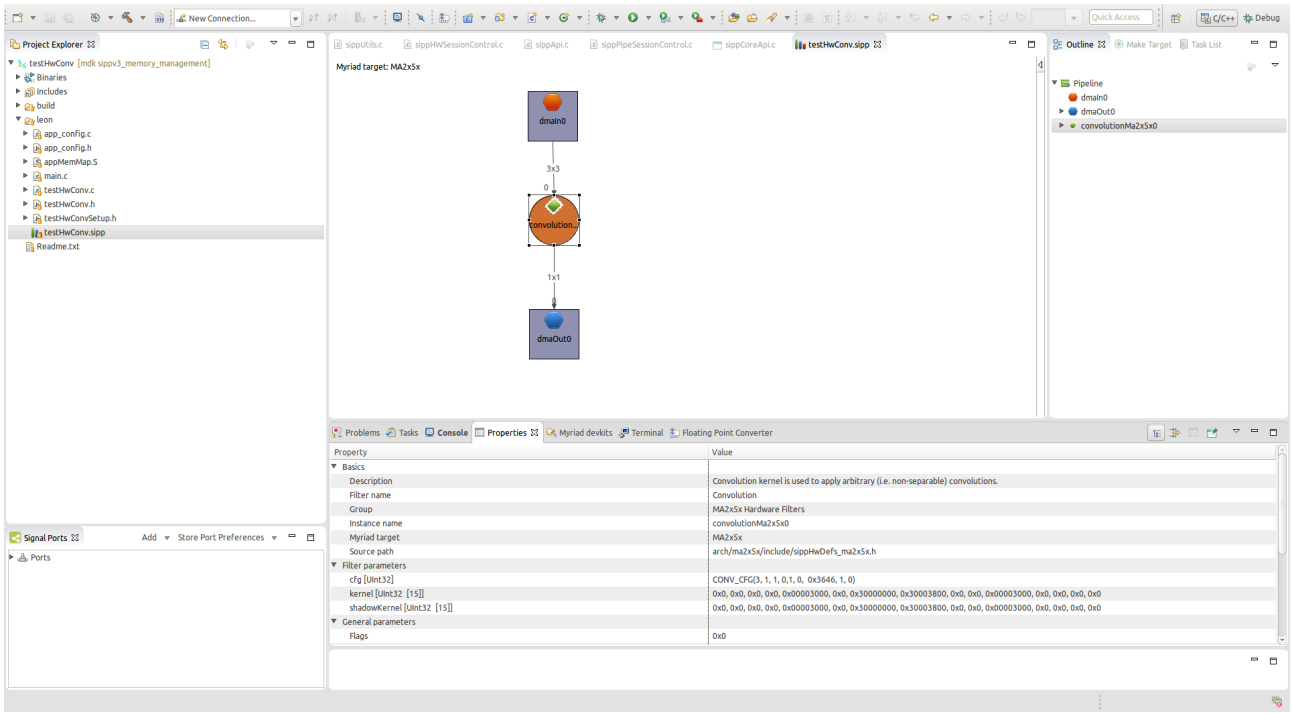


Figure 7: Graph Designer Eclipse Plug-in

The remaining sections describe internal details of the use of the SIPP API and programming.

You must include the *sipp.h* header file in order to use the SIPP API. You can use one of the sample applications, described in the “getting started” section, as a starting point for your application. Any SIPP application must perform the following steps:

Perform any system initialization, as per the example applications. Any application using the MDK must perform this step.

- Instantiate a SIPP pipeline, by calling *sippCreatePipeline().*
- Instantiate some SIPP filters, by calling *sippCreateFilter().*
- Link the filters together to form a graph, by calling *sippLinkFilter().*

Once the steps above have been completed, your SIPP pipeline is ready to star t processing frames of data, by calling *sippProcessFrame() / sippProcessFrameNB().*

3.2 Configuring filters

Filters may have zero or more configurable parameters. The parameters are specified via a filter-specific parameter structure. For SW filters, the size of the configuration structure, in bytes, must be specified via the *paramsAlloc* parameter to *sippCreateFilter().* For HW filters, the struct size is selected from a list of supported HW filter param sizes. If a value other than zero is specified, the SIPP framework allocates the parameter structure internally. The pointer returned by *sippCreateFilter()*points to a structure which has a field named "*params*". This field is a pointer which points to the filter's internally allocated parameter structure. The layout of the actual parameter structure is defined in the filter-specific header file. For example, for the Random Noise addition filter, the parameter structure is named *RandNoiseParam,* and is defined in *<filters/randNoise/randNoise.h>.* The Random Noise filter's *strength* parameter could be configured as follows:

```
RandomNoiseParam *param;

noiseFilter = sippCreateFilter(pl, 0, 640, 480, N_PL(1), SZ(half),
                               SZ(RandNoiseParam),
                               SVU_SYM(svuGenNoise),
                               "Random_Noise");

param = (RandomNoiseParam *)noiseFilter->params;
param->strength = 0.08;
```

NOTE: Parameter modifications take effect at the start of a frame. Hence they are typically updated in between calls to sippProcessFrame() / sippProcessFrameNB().

3.3 Pipeline examples

3.3.1 Simple pipeline

In this example we show how to build a SIPP pipeline with a single filter. We will use the 3x3 Convolution filter as the filter in our example. This filter is included in the MDK. It takes 3 input lines and generates one output line per invocation.

We are going to build a test application which builds and executes the pipeline. In this example our pipeline contains a) a DMA input filter to provide data for the filter, b) the 3x3 convolution filter and c) a DMA out to consume the lines. The DMA in and DMA out filters are built into the SIPP framework.

- Create the pipeline object.
- Create the filter objects.
- Connect the inputs and outputs to create the filter:

```
#define IMG_W 640
#define IMG_H 480

void appBuildPipeline()
{
   pl = sippCreatePipeline(2, 5, SIPP_MBIN(mbinImgSipp));
```

```
    dmaIn = sippCreateFilter(pl, 0x00, IMG_W, IMG_H, 1,
                             sizeof (u8), sizeof (DmaParam),
                                (FnSvuRun)SIPP_DMA_ID, "DMA_In");
    conv3x3 = sippCreateFilter(pl, 0x00, IMG_W, IMG_H, 1, sizeof (u8),
                                sizeof (Conv3x3Param),
                                SVU_SYM(svuConv3x3), "Conv_3x3");
    dmaOut = sippCreateFilter(pl, 0x00, IMG_W, IMG_H, 1, sizeof (u8),
                              sizeof (DmaParam),
                              (FnSvuRun)SIPP_DMA_ID, "DMA_Out");

    sippLinkFilter(conv3x3, dmaIn,3, 3);
    sippLinkFilter(dmaOut, conv3x3, 1, 1);
}
```

We then write a function which processes a frame through the pipeline. This function first initializes each filter's parameter data-structures before it executes the pipeline to process an entire frame.

```
void appProcFrame(SippPipeline *pl)
{
    DmaParam *dmaInCfg= (DmaParam*)dmaIn->params;
    DmaParam *dmaOutCfg = (DmaParam*)dmaOut->params;
    Conv3x3Param *convCfg = (Conv3x3Param*)conv3x3->params;

    dmaInCfg->ddrAddr = (u32)&iBuf;
    dmaOutCfg->ddrAddr = (u32)&oBuf;
    convCfg->cMat[0] = cMat[0];
    convCfg->cMat[1] = cMat[1];
    convCfg->cMat[2] = cMat[2];
    convCfg->cMat[3] = cMat[3];
    convCfg->cMat[4] = cMat[4];
    convCfg->cMat[5] = cMat[5];
    convCfg->cMat[6] = cMat[6];
    convCfg->cMat[7] = cMat[7];
    convCfg->cMat[8] = cMat[8];

    sippProcessFrame(pl);
}
```

The main body of the test application calls the functions defined above in order to prepare and run the pipeline. It uses some SIPP helper functions to read and write image files.

```
int main()
{
    sippPlatformInit(); // SIPP infrastructure initialization

    // Read a frame from file
    sippRdFileU8(iBuf, IMG_W*IMG_H, ""lena_512x512_luma.raw");

    // Build the pipeline and process the frame through it
    appBuildPipeline();
    appProcFrame(pl);

    // Dump the generated output
    sippWrFileU8(oBuf, IMG_W*IMG_H, "lena_512x512_RGB_conv3x3.raw");
    return 0;
```

```
}
```

3.3.2 Simple application using asynchronous API

Using the example of 3.3.1 as a base, the application may be easily converted to use the non blocking API `sippProcessFrameNB()`.

```
void appProcFrame(SippPipeline *pl)
{
   DmaParam *dmaInCfg= (DmaParam*)dmaIn->params;
   DmaParam *dmaOutCfg = (DmaParam*)dmaOut->params;
   Conv3x3Param *convCfg = (Conv3x3Param*)conv3x3->params;

   dmaInCfg->ddrAddr = (u32)&iBuf;
   dmaOutCfg->ddrAddr = (u32)&oBuf;
   convCfg->cMat[0] = cMat[0];
   convCfg->cMat[1] = cMat[1];
   convCfg->cMat[2] = cMat[2];
   convCfg->cMat[3] = cMat[3];
   convCfg->cMat[4] = cMat[4];
   convCfg->cMat[5] = cMat[5];
   convCfg->cMat[6] = cMat[6];
   convCfg->cMat[7] = cMat[7];
   convCfg->cMat[8] = cMat[8];

            sippProcessFrameNB(pl);
}

void appSippCallback ( SippPipeline *             pPipeline,
                       eSIPP_PIPELINE_EVENT       eEvent,
                       SIPP_PIPELINE_EVENT_DATA * ptEventData
                     )
{
    if (eEvent == eSIPP_PIPELINE_FRAME_DONE)
    {
        printf ("appSippCallback : Frame done event received : \n");
        testComplete = 1;
    }
}

volatile u32  testComplete = 0x0;

int main (int argc, char *argv[])
{
    appBuildPipeline();
    ...

    // Register callback for async API
    sippRegisterEventCallback (pl,
                               appSippCallback);

    ...

    appProcFrame(pl);

    ...
```

```
    while ( testComplete == 0x0 )
    {

    }

   // Dump the generated output
   sippWrFileU8(oBuf, IMG_W*IMG_H, "lena_512x512_RGB_conv3x3.raw");
   return 0;
}
```

3.4 Configuring SIPP

3.4.1 Run-time execution

3.4.1.1 CMX Pool size

The user can control the size of SIPP CMX-memory pool. The default size is 32 Kb (as defined in sipp/arch/ma2x5x/build/sippMyriad2Elf.mk).

The user can override the default pool size with the Makefile options e.g.:

```
CCOPT += -D'SIPP_CMX_POOL_SZ=32768'
```

3.4.1.2 Mutex

For Shave synchronization purposes, SIPP component uses 2 mutexes. Immediately after pipeline creation (via sippCreatePipeline), these two SIPP mutexes map on Mutex0 and Mutex1. User however user can re-assign mutexes as in example below:

```
pl = sippCreatePipeline(…);

// After pipe creation – Default : SippMtx0 = Mutex0,
// SippMtx1=Mutex1
// Reassignment after pipeline creation: SippMtx0 =
// Mutex20, SippMtx1=Mutex21
pl->svuSyncMtx[0] = 20;
pl->svuSyncMtx[1] = 21;
```

Clearly in a multi-pipeline scenario, if numerous pipes employing SHAVEs were running concurrently, each would need its own pair of MUTEXES.

3.4.1.3 DMA Settings

To drive data in and out of DDR, the SIPP uses a CmxDma Linked agent component.

By default, SIPP uses LinkedAgent0 and Interrupt0 from CmxDma block. However, these defaults can be changed via Makefile options by setting values for SIPP_CDMA_AGENT_NO and SIPP_CDMA_INT_NO macros, e.g.

```
CCOPT += - D'SIPP_CDMA_AGENT_NO=1'
CCOPT += - D' SIPP_CDMA_INT_NO=10'
```

4 Memory usage and SIPP

This section describes how SIPP uses CMX and DDR memory. Memory is used at two different places:

1. At pipeline initialization
2. During pipeline execution.

For small pipelines it may be sufficient to use CMX for both initialization and execution but for larger, more complex pipelines DDR can be used.

4.1 Physical Pools

The framework has the concept of 4 physical pools. These are:

- CMX Pool
- DDR Pool
- CMX Scheduler Pool
- CMX Lines Pool

As can be ascertained from the naming convention – three of these pools are located in CMX and the fourth in DDR.

Figure 8 shows how each physical pool is used. An important point to note is that the CMX Pool and DDR pool are shared between all active pipelines whereas the scheduling pool and Line Buffer pool are unique to a single pipeline. They are unique in that they are either displaced at a different physical location or their usage is displaced in time from that physical location's use by another pipeline.

The common pools (CMX Pool and DDR Pool) are managed as heaps in order to enable memory collection at pipeline termination.

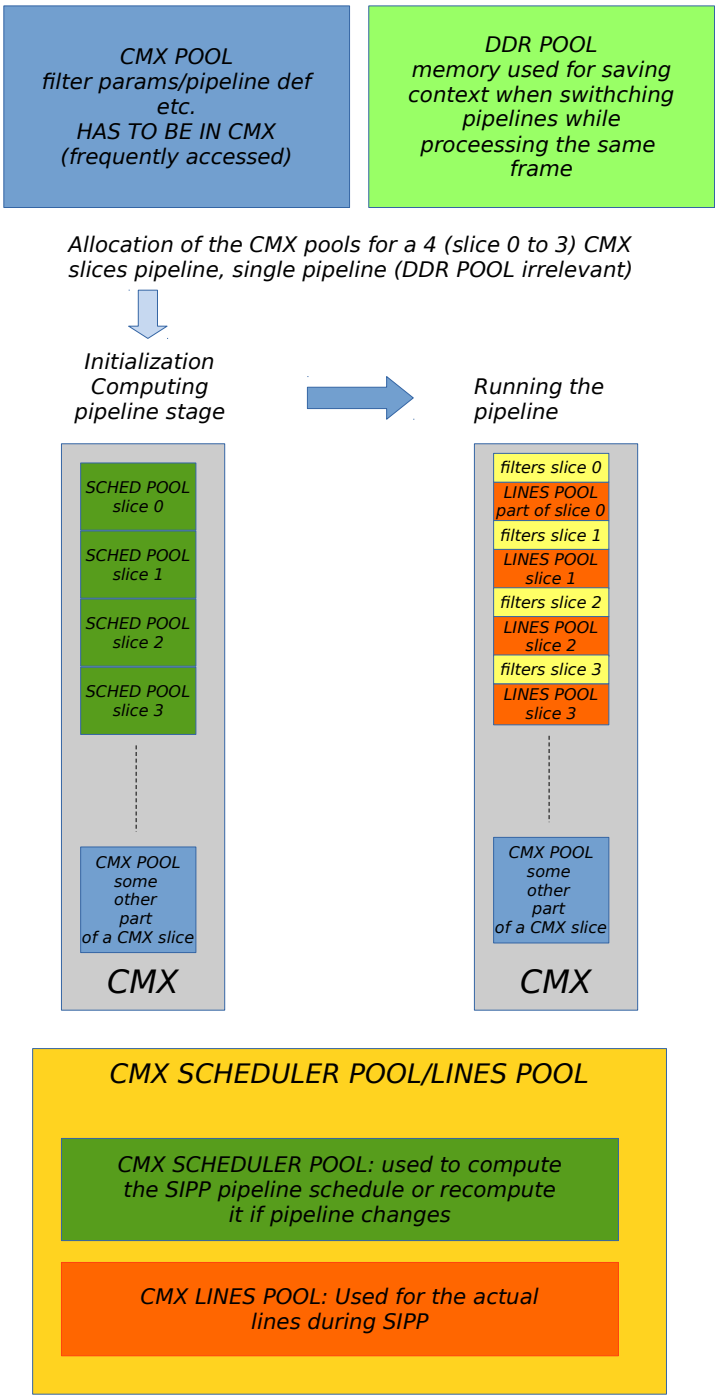


Figure 8: CMX use

4.2 Virtual pool concepts

All memory allocation within the framework itself is serviced via virtual pools. These pools are not specific to any particular physical device. Rather they represent a means of grouping structures which may have a relationship to each other, both in terms of their frequency of use and their lifetime.

Each virtual pool can then be individually mapped on a per stream basis to a physical pool. For the majority

of the virtual pools only a mapping to the physical pools CMX Pool and DDR Pool are recommended. In the case of vPoolCMXDMADesc, only the CMX pool is considered suitable.

Virtual Pool	Default Physical Mapping
vPoolGeneral	DDR pool
vPoolPipeStructs	CMX Pool
vPoolFilterLineBuf	CMX Lines Pool
vPoolCMXDMADesc	CMX Pool
vPoolSchedule	CMX Pool
vPoolScheduleTemp	CMX Scheduler Pool

Table 3: Default virtual to physical pool mappings

A new API, `sippChooseMemPool()` is provided to enable a runtime altering of these mappings on a per stream basis. However, this API only supports remapping to either of the DDR or CMX heap managed pools.

The oPipe runtime detailed in chapter 5.1.5 requires the SIPP framework to allow line buffers to be fully allocated within individual slices. In this case allocation of physical pool descriptors for each slice is needed so that a virtual slice pool can target them, so the vPoolFilterLineBuf0, vPoolFilterLineBuf1..., vPoolFilterLineBuf11 virtual pools were added.

5 Pipeline performance measurement and optimization

The programming paradigm exposed by the SIPP framework deliberately abstracts the client from many of the necessary aspects of establishing and running pipelines. However the task of maximising the potential performance of a pipeline within a particular execution context mandates a degree of understanding of the SIPP pipeline runtime. This understanding in turn offers a means to interpret the metrics which are available and to guide usage of the various tuning levers which are available to increase a pipeline's performance.

5.1 Introduction to SIPP Pipeline runtime

As part of the pipeline finalization process (triggered through a call to one of the sippFinalisePipeline, sippAllocCmxMemRegion, sippProcessFrame or SippProcessFrameNB APIs) the SIPP framework performs a basic piece of pipeline analysis which among other items allows it to select the most suitable scheduler and runtime for the pipeline from a pool of such at the pipelines disposal.

The primary concerns of this selection will be ensuring compatibility with existing pipelines and maximising performance within those criteria. However, the goals of the chosen scheduler and runtime will be to handle the pipeline under their control as efficiently as possible so as to maximise the performance and potential throughout.

This sub-section focuses on the SIPP runtime. It will briefly describe the runtime tasks and also look at the overall subsystem under the control of the runtime to illustrate how performance may be maximized.

The SIPP runtime is distributed across the LEON processor on which the SIPP framework is running – and across any SHAVE processors assigned to executing SW kernels on SIPP pipelines. However, the SHAVE aspect to the SIPP runtime is quite fixed and is not subject to selection by the SIPP framework based on pipeline characteristics as per the LEON SIPP runtime. I fact all LEON SIPP runtimes requiring a SHAVE runtime use the same version. To be clear this section focuses on the LEON SIPP runtime as the SHAVE SIPP runtime does not afford any tuning options and is already considered to be optimal for the execution of SW kernels.

5.1.1 Tasks of LEON SIPP runtime

While the exact nature and list of tasks may differ from pipeline to pipeline depending on various issues (for example does a pipeline have HW filters/ SW filters etc.?) the list following details an overview of the main functionality carried out by the runtime.

- Extract the commands for the current iteration from the pipeline schedule.
- Start all HW filters which need to be started for an iteration.
- Start all SW filters which need to be started for a particular iteration.
- Start all DMA filters which need to be started for current iterations.
- Perform all line preparation tasks for the subsequent iterations – the ultimate output of this process is the generation of addresses for every line of the input kernel(s) and output line(s) of SW and DMA filters (HW filters can manage this task themselves).
- Handle any multiple context HW filters (certain conditions only).
- Establish that all started units have completed.
- Update any HW filter input buffers with information on lines added to the buffers on completion of the current iterations.

- Restart the process for the next iteration!

5.1.2 Synchronous versus asynchronous runtimes

If a runtime is operating in asynchronous mode, after all line preparation tasks have been performed the runtime will quit the execution context in order to await further interrupts informing of the completion of the activities of filters involved in the current iteration. If the runtime is operating in synchronous mode, establishing that all units have completed will involve polling the operational status of the active units until it is ascertained that all are done.

5.1.3 Generic runtime

Generic runtime is the SIPP runtime which is suitable for any type of pipeline the clients may use in the SIPP framework. It is adjusted to handle any valid situation it may encounter. Generic runtime is selected by the framework by default, if the application doesn't have options to use any other runtime. For some pipelines generic runtime may not be very efficient, as it may do lots of things not needed for certain pipelines. So if certain features of the pipeline are recognized, a different runtime may be called, one that has been specially tailored to the features of the pipeline.

Generic runtime is based on one line being output by each filter which is scheduled to run in that iteration and can operate on SW pipelines, HW pipelines or on mixed pipelines.

5.1.4 Multiple Lines Per Iteration runtime

With Generic runtime, a very accurate model of what data will be available from each filter can be kept and the amount of memory used by each filter is kept to a minimum. However, it also means that each filter gets serviced by the SIPP framework on every output line, which means a lot of setup and a lot of additional processing. So the alternative is to ask each filter to produce multiple lines of output every time it is called. This would mean that the processor has to get involved fewer times allowing performance to go up and more cycles on the processor to be made available for use by other applications.

Multiple Lines Per Iteration runtime offers the possibility to each filter of producing more than one line of output every time it is called. MLPI runtime will be invoked whenever the client calls a new API sippPipeSetLinesPerIter() setting the number of lines per iteration to 2/4/8 or 16. The basic premise is that using this runtime, an increase in performance will be achieved which should be traded off against the additional (CMX) memory requirements it brings.

5.1.5 OPipe runtime

For all except one of the included runtimes, the task list as defined in 5.1.1 is correct. However for the sake of completeness this sub-section will highlight the anomalous behaviour of the oPipe runtime when it is selected.

The framework will select the oPipe runtime for a pipeline if that pipeline is composed entirely without SW filters and without HW filters being re-used more than once in the same pipeline. It is chosen because it takes advantage of a HW feature added in MA2150 enabling SIPP HW filters to perform their own flow control and scheduling when given information about the configuration and usage of their input and output buffers. This mode renders the task of creating a schedule meaningless since all filters in the pipeline will in effect be free running. This means their rate is controlled only by the availability of data in the input buffer and space to decode into within their output buffers. The main reason behind adding the oPipe runtime is to enable direct streaming between certain filters in the fixed oPipe configuration, saving a lot of power and bandwidth in this way.

The tasks of the oPipe runtime are as follows

- Commence any input DMA operations.
- On completion of an input DMA operation, inform all filters connected at their inputs to the buffer just filled as to the availability of this new input data.
- Monitor all HW filters for interrupts signalling they have completed processing a certain number of lines. When required, trigger DMA operations to drain these lines from the output buffers of filters by commencing DMA out operations sized at a specified number of lines.
- On completion of an output DMA operation, inform the filter whose output buffer has been read that a specified number of lines has been consumed from the output buffer and this space may not be considered free for re-use by new operations.

To enable the pipeline to take into consideration the oPipe runtime, the macro `SIPP_SCRT_ENABLE_OPIPE` should be defined and the pipeline flag `PLF_CONSIDER_OPIPE_RT` must be set before finalisation of the pipeline (See Table 7 for flag details).

It should be noted that the direct streaming runtime is naturally running using the asynchronous mode. While the other runtimes use the DMA unit assigned to SIPP, the oPipe runtime uses the driver for the input and output DMA operations, so the application must enable CMX DMA driver interrupt handler when using the oPipe runtime before starting to run.

oPipe runtime allows the client to hand-craft the buffer allocation for each filter needed and not use the internal algorithm to decide buffer sizes. The number of lines per output buffer can be set for each filter through a call to sippPipeSetNumLinesPerBuf(). Larger number of lines per output buffer should give a speed-up at runtime at the expense of requiring more line buffer memory.

A further point to note at this stage is that due to its very different nature, not all of the measurements suggested in 5.2 will be available when this runtime is selected.

5.2 Performance measurement

Ultimately with performance measurement a CC/pix (clock-cycle-per-pixel) metric will be used as the metric which will be made available for interpretation of the pipeline performance. This metric will be composed of a measurement of the number of clock cycles which elapse between the call to an API triggering the framework to commence processing of a frame and an indication from the framework that such processing is completed to a degree to which the framework is capable of immediately handling further processing. This value is in turn divided by the number of pixels (in one plane) of the input image to create the final metric.

While this metric is suitable for final performance evaluation, the framework also considers which metrics are of interest in the context of facilitating intelligent optimization. To this end the framework provides a further set of figures which may be used as a guides for which optimization steps detailed in 5.3 may yield best results in terms of increasing performance. Ultimately the aim is also to provide sufficient information to enable the user to decide when optimization could yield significant results and at the other end of the spectrum to recognise when a performance is already sufficiently optimal to render meaningful return from further optimization as difficult and potentially unobtainable.

5.2.1 HW pipelines

On HW pipelines side, two examples have been added: MonoPipe and ISPPipe, that can be found in

mdk/examples/Sipp/SippFw/Runtimes.

Both of tests are using only HW filters and they were used for performance measurements. The metrics that will be used for the interpretation of the pipelines performance are clock cycles per pixel (cc/pix) and memory footprint.

5.2.1.1 Clock cycle per pixel

The value for the first metric needs to be minimized in order to increase the performance of the pipeline. The following figures illustrate the values for the clock cycles/pixel obtained when the tests are run with different runtimes and different resolutions. In these particular cases performance measurements are made using generic runtime, MLPI runtime at 4 lines per iterations and the oPipe runtime. The tests are run over three resolutions: VGA, HD and 4K. It should also be noted that all tests use asynchronous mode.

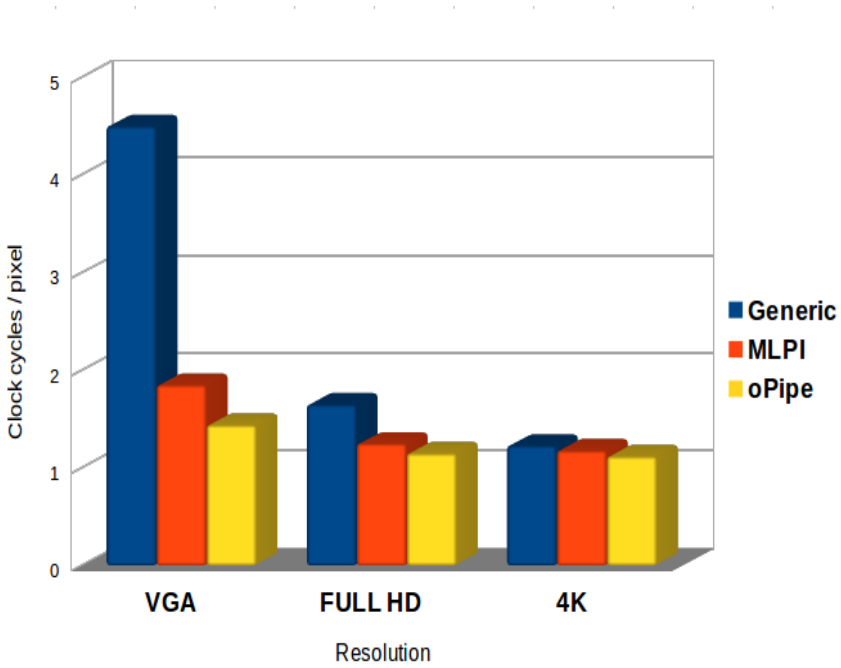


Figure 9: MonoPipe

Figure 9 shows the cc/pix values for the MonoPipe test. It can be seen that as horizontal resolution increases, the performance difference between runtimes decreases.

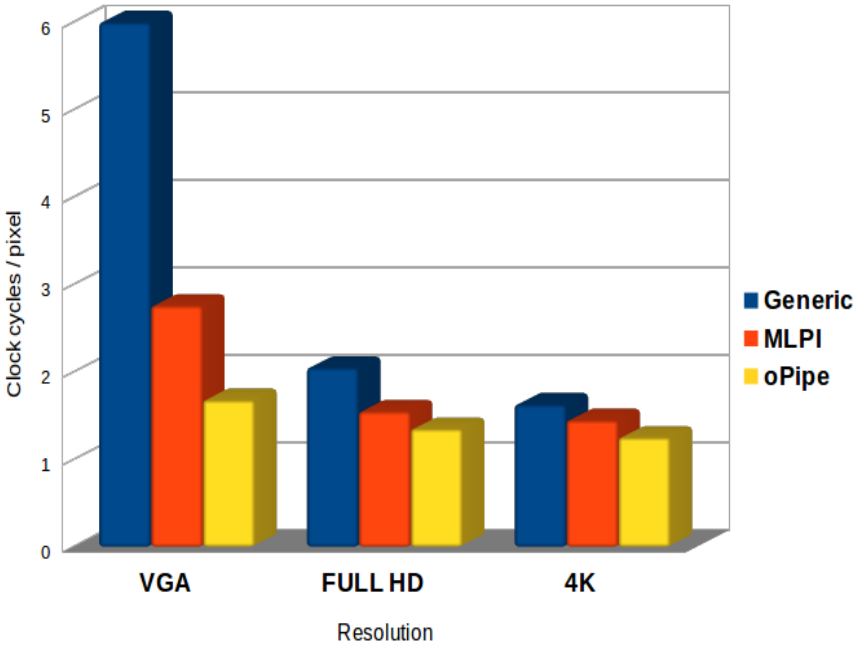


Figure 10: ISP Pipe

Figure 10 shows the cc/pix values for ISPPipe test. As in the previous case, the runtime and the resolution that are used in the test will have a major impact on performance. The best case of improving performance for FULL HD resolution is when oPipe runtime is chosen. Note for example that at HD resolution, cc/pix is about 36% reduced for the oPipe runtime when compared to the generic runtime.

5.2.1.2 Memory footprint

The second metric which will be discussed regarding the pipeline performance is the memory footprint. This is in turn broken into two main areas:

- Framework structure allocation.
- Line buffer allocation.

Framework heap usage – in terms of the framework heap, the space is decreased a lot especially when oPipe runtime is used (see Figure 11).

For example when the test is at 4K resolution, the CMX pool size decreases by 61% if MLPI runtime is chosen instead of generic runtime. If the oPipe runtime is used it will be saved almost 80% of total space occupied when generic runtime is used. A large percentage of this saving relates to the schedule. When MLPI is used, the schedule reduces in size by an order approaching the number of lines per iteration set. For example, with 4 lines per iteration, the schedule size may be approximately one quarter that of the generic runtime. For the oPipe runtime no schedule is required.

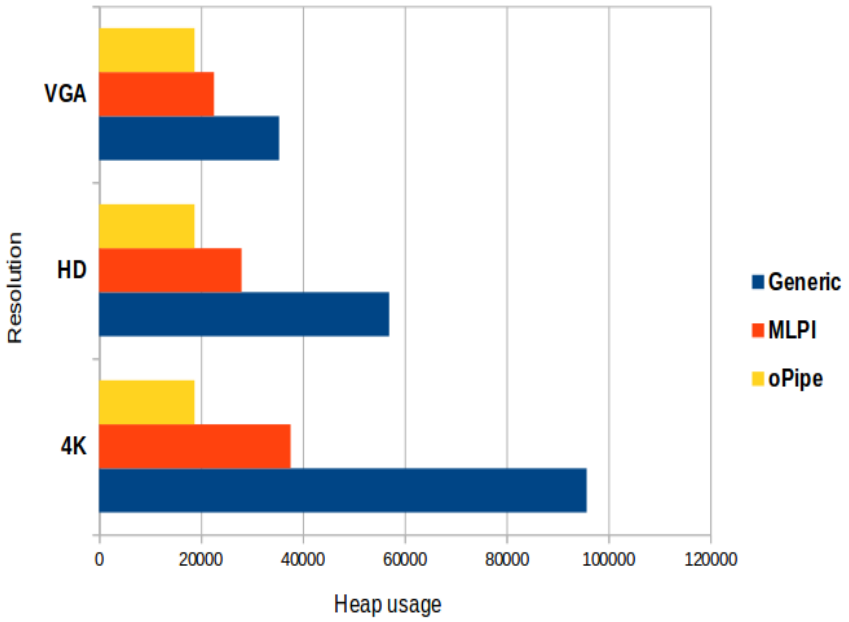


Figure 11: Framework heap usage

Line buffer allocation – considering the line buffer usage for generic and multiple lines per iteration all line buffers are allocated from vPoolFilterLineBuf. The actual usage for line buffer pool is given by the memory used in that pool multiplied by the number of slices allocated to the pipeline.

In the following illustrations we take for example the MonoPipe test that uses an image with a VGA resolution and we can notice, depending on the runtime used, generic or four lines per iteration, how much vPoolLineBuf is occupied.

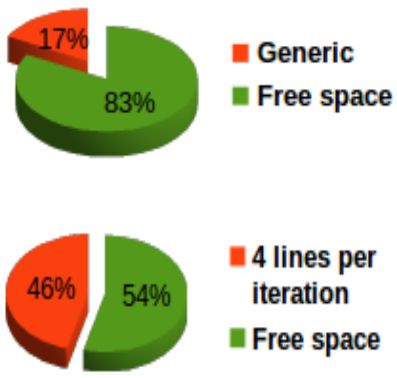


Figure 12: Line buffer usage

Regarding the oPipe runtime, we create a set of virtual pools one for each slice that is allocated to the pipeline. In this case, the actual memory usage will be the amount that is used in each virtual pool.

For example, the MonoPipe test is running on 4 slices. The pools allocated will be the first four: vPoolFilterLineBuf0, vPoolFilterLineBuf1, vPoolFilterLineBuf2 and vPoolFilterLineBuf3.

The measurements for the line buffer were made in the same situations presented for the other metrics. It could be noticed in most of the time that the line buffer usage was almost three times increased when we used 4 lines per iteration and more than four times when we used oPipe runtime against generic runtime.

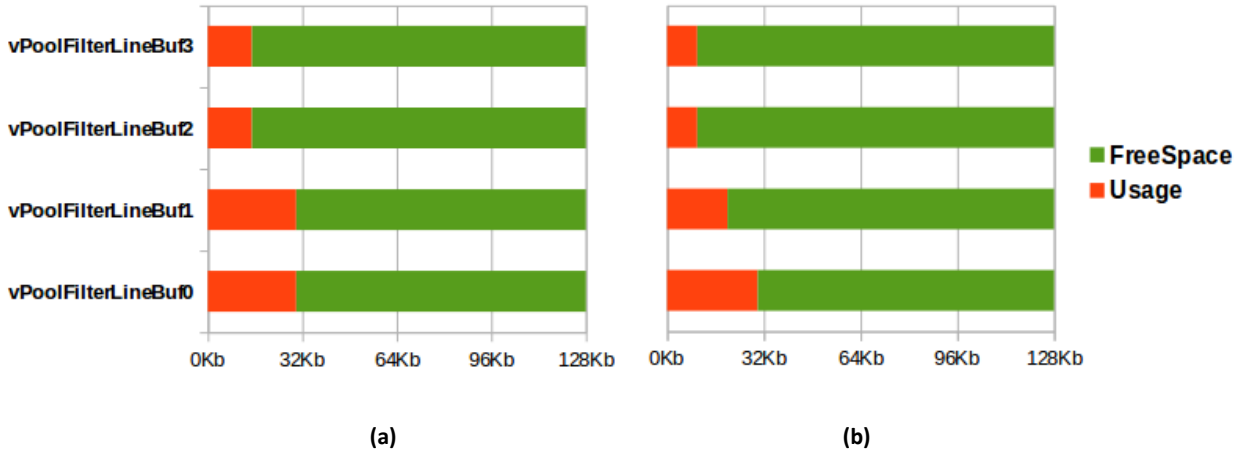


Figure 13: Line buffer usage for oPipe runtime

Figure 13 shows the difference of line buffer usage in oPipe runtime when using the internal algorithm for buffer allocation (a) and when the client sets up the buffer sizes (b).

Running in oPipe runtime not only increases performance, but it also reduces the total number of interrupts taken, which can be a critical factor.

5.2.2 Enabling run time statistics gathering

There are two steps to enabling runtime statistic gathering. These break down into a build time and a runtime aspect.

To enable statistics gathering at build time, the macro `SIPP_RUNTIME_PROFILING` should be defined (within `sippMyriad2Elf.mk`). There is a build time component because even minimal runtime checks to ascertain if statistics should be gathered consume some clock cycles and processing MIPS and therefore the option to remove them entirely to achieve maximum performance is given.

Enabling the statistics gathering at runtime for a pipeline is achieved by setting the pipeline flag `PLF_PROVIDE_RT_STATS` before finalisation of the pipeline (See Table 7 for flag details).

All measurements are taken using the system clock so the statistics gathering does does not need a specific timer.

It should be noted that while minimally invasive – run time gathering of statistics may well have a slight adverse effect on performance. The statistics are gathered into memory assigned from the `vPoolGeneral` virtual pool which by default is mapped to DDR (see Table 3) – clearly the physical mapping of this pool to DDR or CMX may have some further minor impact on how invasive the statistics gathering is in terms of performance. The size of the area is controlled by two build time macros defined in sippCfg.h – `SIPP_RT_ITER_STATS_SIZE` which dictates how many iterations will be covered (should be greater than or equal to the number of iterations calculated by the scheduler) and `SIPP_RT_PER_ITER_STATS_SIZE`

which dictates the maximum number of measurements which can be recorded on each iteration.

Clearly the user is free to move around the measurements points in order to record different metrics for each iteration. The subsequent sub-sections detail some of the metrics which may be targeted.

5.2.3 Scheduling time

Of most interest in situations where rescheduling is frequent such as pyramid scaling. This measurement is always available in the run time statistics. It is updated only once, when the initial schedule is created – further calls to `sippReschedulePipeline()` do not update it.

When an oPipe scheduler / runtime is selected, this metric has little merit as oPipe does not use a pre-crafted schedule.

5.2.4 Iteration processing times

Each iteration of the schedule is in effect a separate and distinct step for the runtime in the processing of a frame. Iterations run sequentially in time. Within each iteration there are 4 main activities, each of which there is merit in timing. These are

- Operation of HW filters.
- Operation of SW filters.
- Operation of DMA units.
- Framework preparation for next iteration.

The timing of each of these elements helps identify the “long-pole” on each iteration. That is to say it points to which of the 4 elements should be focused on first in order to improve the performance figure.

Note – the accuracy of such measurement may not be as accurate in asynchronous mode due to inherent addition of interrupt latency into the degree of error. However, the results may still be used to provide a clue as to what is the long-pole. If more accuracy is sought, there may be merit to switch to the synchronous mode using the blocking API if the system under test is composed of a single pipe or may be broken down to single pipes for the purposes of optimization.

5.3 Optimization

To be clear this section is chiefly interested in pipeline optimization for performance. That is to minimise the cc/pix value described in 5.2. Optimization against any other metric such as memory footprint is not dealt with. However the sub-sections which follow will highlight relationships between certain resources and performance which may also be used to release resource from a pipeline which is measured to achieve a performance beyond that which is required in order that some other aspect may use this resource to have its performance optimised.

HW filters are often specified for 1 cc/pix throughput (though not guaranteed for all data types in all filters). The maximum theoretical throughput of SW kernels may also be calculated. Therefore it should be possible to calculate a theoretical value of clock cycles per pixel for a pipeline. Once the pipeline has been run and measured, it may be considered that a large deviation between measured and theoretical values will produce a desire to optimise the pipeline configuration.

This sub-section presents a set of optimization options aimed at providing the user with a simple pathway to achieve the desired performance, or to identify when performance is already approaching the optimal for the system.

5.3.1 Optimisation techniques

The following sub-sections detail some techniques which may be used to improve the measured performance of a pipeline. While they work under various circumstances, these techniques will be referred from section 5.3.2 to provide an effective guide to the types of situation for which each technique is best suited.

5.3.1.1 HW CMX slice start offsets

When more than one slice is assigned to a pipeline for use in the allocation of filter line buffers for that pipeline – the lines will be partitioned into a number of chunks equivalent to the number of slices allocated such that some of the line exists in each of the allocated slices. SIPP HW filters and the SIPP framework support the offsetting the start slices of the HW filters to minimise CMX contention

So for example, consider a HW pipeline involving a DMA In → LUT filter → convolution filter → scalar → DMA out. 4 CMX slices are assigned to the pipeline at sippCreatePipeline() time. The pipeline may well have 4 filter buffers, located at the outputs of the DMA In, LUT, Convolution and Scalar filters. By default the first pixel of the output lines of each filter will be placed at the start of the first chunk of the line and the first chunk will be in the first slice assigned to the pipeline. This means that when all filters are started simultaneously, they will start fetching their input data from the same CMX slice, leading to contention on that slice. By simply offsetting the slice start of filters as per table

Filter	Output Buffer First Slice
DMA In	0
LUT	1
Convolution	2
Scalar	0

Table 4: Example filter output buffer slice starts

Note that no output filter has its buffer start in slice 3. This is because only HW filters support this offsetting at both input and output. While the Scalar filter would have been capable of writing to an output buffer with an offset start slice, the DMA Out filter would have been incapable of consuming it. So care must be taken that candidate filters for this are correctly chosen. To be clear, this is only supported for HW filters and even then only when their output is consumed only by other HW filters and not SW filters or CMX DMA units.

Offsetting the start addresses is not an automatic process within the framework. Support is limited to the provision of the hooks enabling the offset being exposed through the SippFilter structure so that the client may enforce this if necessary. For example, in order to create the first slice mappings expressed in Table 4, the following code may be used:

```
dmaIn       = sippCreateFilter(...
lut         = sippCreateFilter(...
convolution = sippCreateFilter(...

...
```

```
dmaIn->firstOutSlc       = 0;
lut->firstOutSlc         = 1;
convolution->firstOutSlc = 2;

sippFinalisePipe(...
sippProcessFrame(...
```

Note that as per all configuration of filters, this should be done before any call to `sippFinalisePipe()`, `sippProcessFrame()` or `sippProcessFrameNb()` to ensure correct results.

5.3.1.2 Use of SW command queues

Starting the CMX DMA units and in certain runtimes, updating and starting the HW filters can be a non-trivial task. As opposed to simply writing pre-calculated values to fixed addresses, the values to be written are calculated at runtime as opposed to in the scheduler in order to save space (since the schedule is often kept in CMX and potentially a very significant number of values per iteration need to be kept this is considered the preferred approach).

The runtime calculation of these values normally occurs within the start and end of iteration framework overhead (illustrated in Figure 14) so that the values are available at point of use and require no storage. However, the values can be calculated at any point in time. So a compromise is to calculate the values during the framework operations section (as illustrated in Figure 14) so that storage is only required for one iteration's worth of transactions. This results in the start and end of iteration periods decreasing, while the framework operations period increases. In fact the framework operations period will increase by an amount greater than the combined saving in the start and end of iteration periods, so the total MIPS required by the processor has in fact increased slightly. However, if the framework is not the long-pole in the system (see section 5.3.2.1) then this will still result in a potential improvement in the overall system performance as the savings in start and end of iteration periods will count towards the overall iteration time, while the increase in the framework operations period will be masked by the long-pole of the system.

While this mechanism often shows improved results, care should be taken whenever

- The framework is close to being the long pole.
- When running concurrent pipelines with asynchronous API it is desirable for the framework operations period to be kept as short as possible so that other operations (such as starting an alternative pipeline) have a chance to proceed. This aim may outweigh the individual gains seen on a single pipeline. See section 5.3.3 for more details.

5.3.1.3 Increase number of lines per iteration

By default, schedulers and runtimes (with the exception of the oPipe version) will work on the basis of one line of output being produced by each of the filters scheduled to run in each iteration.

Through a call to sippPipeSetLinesPerIter() this value may be changed to 2/4/8 or 16. The functionality exposed through this API represents an attempt to mitigate the effects of the necessary per iteration overhead in both the SIPP framework running on the LEON processor on Myriad2 and the SIPP runtimes running on each SHAVE processor assigned to the pipeline.

Within the SIPP framework, the per iteration overhead consists of

- Start of iteration overhead (see Figure 14).
- End of iteration overhead (see Figure 14).
- Framework operations (see Figure 14).
- For asynchronous mode, interrupt latency + irq handler overhead.

For the SIPP runtime on the shaves, the per iteration overhead consists of

- IPC operations with LEON.
- Accessing padding information per SW kernel.
- Accessing number of plane information per SW kernel.
- Loading lists of filter wrapper functions.
- Loading I/O information for each filter.

It is clearly observed that when a pipeline is run with more then one line per iteration, the iteration overhead increases, but the number of iterations decreases by a larger amount. For example, in moving from one to two lines per iteration, the number of iterations in the generated schedule tends to decrease by almost 50% for an HD image. That is to say there are about half as many iterations. While the per iteration overhead grows, it grows by much less than the 100% value which would wipe out any gain from having 50% fewer iterations. Therefore the total iteration overhead decreases.

It should also be noted that in asynchronous mode, the interrupt latency + irq handle overhead does directly scale relative to the number of iterations – so running more lines per iteration almost always leads to benefit when running in asynchronous mode.

It is observed that this provides a performance increase in most scenarios. However the increases are most significant when Framework operations are considered the long-pole (see section 5.3.2). It is often possible to remove the framework operations from the critical path using this method.

While the advantages of increasing number of lines per iteration have been made clear, the gains need to be weighed against the increased line buffer requirement of the filters when running more than one line per iteration. The increase in line buffer requirement per filter is difficult to predict in advance and experimentation may be required to derive what the requirements are at each setting. It is worth stating here for clarity that the increase is not a direct scale with the increase in number of lines per iteration.

NOTE: In a situation in which the oPipe scheduler and runtime is chosen, the number of lines per iteration is effectively meaningless as it does not employ a mechanism to create a schedule for a certain number of iterations.

5.3.1.4 Write a bespoke runtime

While the framework will attempt to choose the most suitable runtime for the described pipeline, the runtime will still be generic to a degree. It may be possible to tune the runtime for the needs of a specific pipeline. If this is considered advantageous, then the user may provide their own version of the runtime and assign it to the pipeline at the correct point.

Uses of this mechanism include but are not limited to:

- Remove unnecessary checks – for example if a pipeline has no HW filters there is no need to check which HW filters to start in nay iteration which the generic runtime will do.
- Switch the order in which units are started – if it is felt that SW filters will always be on the critical path, ensure these are started first before DMAs or HW filters.

The run-time is overloaded simply by calling sippFinalizePipeline() and then in client code modifying the pipeline member

```
sippRuntimeFunc                pfnSippRuntime;
```

to point at the function implementing the bespoke runtime.

5.3.2 Long-pole optimization

Section 5.2.4 describes the 4 main activities which occur during each iteration under the control of the selected SIPP runtime (excluding the oPipe runtime). In order to begin optimization intelligently, a picture of the relative processing time required for each of these activities proves useful in determining the optimization steps.

NOTE: Before considering how to interpret the runtime statistics in order to ascertain the long-pole, it is worthwhile to reiterate that the user is responsible for placing the instrumentation code in the suitable positions within code to gather information relating to the start and stop time of each of the activities mentioned.

A typical pipeline iteration time-line is illustrated in Figure 14.

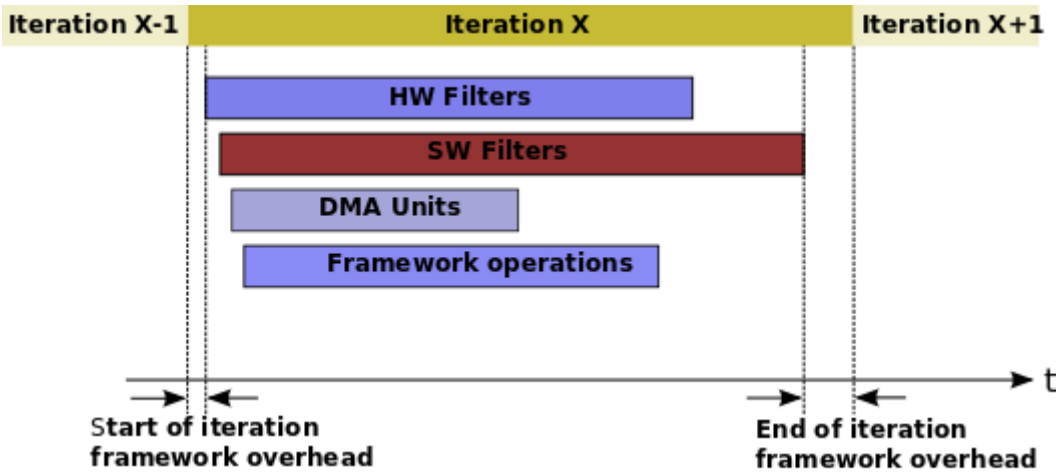


Figure 14: Time-line of per iteration SIPP runtime operations

The iteration begins, immediately after the previous iteration is considered complete. After some cycles one of the 3 functional unit groupings (SW filter, HW filters or CMX DMAs) will have been started. Soon in relatively quick succession the other two groupings will be started (if required on that iteration). Once all required units are operational the framework will dedicate itself to performing the tasks it needs to in order to be ready to start the subsequent iteration.

This methodology ensures that all 4 activities occur mostly in parallel. It is also worth noting that there is no data-dependency between any filters on any iteration. That is to say all filters can run to completion for an iteration independently of the progress of any other filters in that iteration. In turn this means that HW filter progress within an iteration is not gated by SW filter progress or DMA progress and the same is true for the SW filter and DMA activities.

Points to note in Figure 14 include the two framework overhead periods at the start and end of the iteration. Great care has been taken within the runtimes to minimise these periods, as in effect no progress is being by the filter kernels during this time so their durations directly impact the efficiency of the system. It should also be noted that the illustration is not necessarily to relative scale.

Finally there follows a brief and non-exhaustive guide to the factors which influence the time required for

each of the 4 activities:

HW filter time – all HW filters may be run and started together so the total HW time is based on the worst case HW filter on any iteration. This may be based on the specification and configuration of the HW filters, the memory bandwidth, DDR usage of the system as a whole, filter line width etc.

SW filter time – The number of SW filters, the bandwidth and complexity of each of the SW filters, the number of SHAVES allocated to the pipeline, the location of SHAVE code and data, the amount of padding which must be done as well as the general factors of CMX bandwidth and the filter line width

CMX DMA time – The number of DMA operations required in any iteration and the line width of each operation as well as CMX and DDR bandwidths to a large extent dictate the CMX DMA activity time

Framework – Framework activities chiefly employ maintaining buffer models for all buffers used as inputs to SW filters and / or CMX DMA units. These models enable the framework to calculate addresses for each line of the input kernel and for the output lines for each SW or DMA filter. Essentially the effort required here depends on the number of SW and DMA filters in the pipeline and is invariant to line width, data-types etc., unlike all the other activities.

Note also that the end of iteration framework overhead can involve a few tasks among which informing any HW filters of new data within their input buffer is most time consuming.

5.3.2.1 Identification of the long-pole

With run time statistics enabled, timing information should be gathered at appropriate places from the selected runtime in synchronous or asynchronous mode to allow the time required for the 4 activities to be measured. At the end of the frame – these measurements may be analysed to ascertain which of the 4 functional units is most often the bottleneck during a single frame execution on a pipeline.

Gathering statistics to enable runtime long-pole detection is dealt with directly in the detailed example within 5.4. Once the statistics are gathered and the long-pole identified the following sub-sections may be used to guide an optimization plan.

5.3.2.2 HW filters as long pole

When HW filters are the bottleneck of the system there is one standard optimization technique. If more than one CMX slice has been allocated to the pipeline – it is worth offsetting the start slices of the HW filters to minimise CMX contention as described in 5.3.1.1.

Other questions to ask include

- Are all HW filter line buffers located in CMX? If not could they be moved there?
- Is the framework sufficiently far from being the long pole that the use of the SW command queues described in 5.3.1.2 could be considered? If the framework then becomes the long pole, using multiple lines per iteration also could see the framework be removed from the critical path once more.
- Does the selected runtime start HW filters first, before SW filters and DMAs? If not this switch could be made in code.
- If running using the asynchronous API, using multiple lines per iteration as per section 5.3.1.3 should always have a strong chance to show some improvement.

If these techniques are tried and show no improvement, it is unlikely the performance can be significantly further improved even using the most bespoke tuning.

Finally, it is worth noting that if after optimization HW remains the long-pole in a pipeline including SW filters, it may be worth considering using fewer shaves. Since HW is the long-pole, it may be possible to use

fewer shaves and still not seriously effect the overall system performance while releasing this valuable resource for use in an alternative activity.

5.3.2.3 SW filters as long pole

Attempts to accelerate a pipeline in which SW filters are the long pole should consider the following

- Can more SHAVEs be added to the pipeline.
- If running using the asynchronous API, using multiple lines per iteration as per section 5.3.1.3 should always have a strong chance to show some improvement.
- Multiple lines per API has the possibility of showing improvement even with the synchronous API by reducing the SHAVE runtime overhead.
- Does the selected runtime start SW filters first, before HW filters and DMAs? If not this switch could be made in code.

It is worth noting that the rate of return seen in adding additional SHAVES to pipelines running SW kernels drops off as more SHAVES are assigned. The reason is the general efficiency of the SW processing drops off the more shaves which are added. The reason for this is the more SHAVEs which are added, the shorter the section of line which each shave will cover. However, the per line and the per iteratyi0on overhead remains the same, so effectively the time spent doing useful work drops while the time spent dealing with overhead on each SHAVE remains constant. So the efficiency drops off as the time spent doing useful work drops as a percentage of the total. Of-course in absolute terms, the total time spent executing SW tasks should drop when more SHAVES are added to the pipeline.

5.3.2.4 CMX DMA as long pole

- Is the framework sufficiently far from being the long pole that the use of the SW command queues described in 5.3.1.2 could be considered? If the framework then becomes the long pole, using multiple lines per iteration also could see the framework be removed from the critical path once more. SW command queues can speed up the process of starting the DMAs.
- Does the selected runtime start DMA filters first, before HW filters and SW filters? If not this switch could be made in code.
- If running using the asynchronous API, using multiple lines per iteration as per section 5.3.1.3 should always have a strong chance to show some improvement.

5.3.2.5 SIPP framework as long pole

In a situation in which the SIPP framework is the long pole in a pipeline execution the main option available is to attempt to run more than one line per iteration as described in 5.3.1.3.

The SIPP framework tends to be on the critical path for 'fast' pipelines. That is to say pipelines which have one or more of the following characteristics.

- Use mainly HW filters.
- Have small line widths.
- Assign many SHAVES.
- Run simple SW kernels.

The time taken for framework operations is independent of line width, filter complexity etc. So the same pipeline at HD may have SW filters as a long-pole, but at VGA has framework operations as its long pole.

Running more than one line per iteration should scale the HW and SW operations time in direct proportion,

while scaling the framework operations time at a lesser percentage, therefore bringing the framework operations time closer to the others or potentially even removing it from the critical path altogether.

Finally, it is worth noting that if after optimization framework operations remains the long-pole in a pipeline including SW filters, it may be worth considering using fewer shaves. Since framework operations is the long-pole, it may be possible to use fewer shaves and still not seriously effect the overall system performance while releasing this valuable resource for use in an alternative activity.

5.3.3 Multi-pipeline optimization

In order for the SIPP framework to run multiple pipelines in parallel, the first consideration is that the pipelines must not compete for shared resources. That is to say, HW filters may be assigned only to one pipeline at any time and SHAVE units will also be assigned specifically to one pipeline at any time. However should pipelines use different SHAVE groups and different HW filters, there is no reason for them not to be run in parallel.

A secondary and less intuitive consideration is that the pipeline performance must enable pipelines to run in parallel!

With the current 'bare-metal' implementation of the SIPP framework, a client may make two consecutive calls to `sippProcessFrameNB()` in their application in the following way:

```
sippProcessFrameNb (pPipe0);
sippProcessFrameNb (pPipe1);
```

It may be reasonably expected that should the criteria for not competing on resource be satisfied then the two pipelines could run in parallel. However, when running in asynchronous mode, the runtime executes within interrupt context. It is possible that when pPipe0 is started by the first call to `sippProcessFrameNb`, interrupts may begin to be taken before. The second call to `sippProcessFrameNb` has an opportunity to pass the API request for pPipe1 to the resource scheduler. If the interrupts are sufficiently close together it could be that the insufficient processor MIPS are available outside of interrupt context during execution of the entire frame for pPipe0 in order to actually get pPipe1 started in order that it may run in parallel.

In this situation, the performance of the first pipe is almost too good for the second pipe to be able to run in parallel. The net effect is that the pipelines run in series and end up taking more combined cycles than they should. A further effect is that the advantages of a non-blocking API are in effect lost since no real work may take place outside of interrupt context while the pipeline is running.

This situation is demonstrated in the example described in section 5.4.

5.4 Performance example guide

With the 16.06 release of the Movidius MDK, a new example has been provided to enable performance measurement and analysis of SIPP pipelines. It is located in

```
/mdk/testApps/components/sipptests/Performance/sippPerfTestBed
```

The leon/main.c file provides the testbed within which pipelines may be created and measured. It further contains display and logging capability for result presentation. The file leon/perfTestbedPipes.c contains example pipelines.

The application is built using build/myriad2/Makefile. This Makefile contains the switch

```
CCOPT    += -DSIPP_RUNTIME_PROFILING
```

to enable runtime profiling within the SIPP framework build. Profiling is then available to be activated at runtime. This is achieved via the lines

```
pipe->sippFwHandle->flags |= (PLF_PROVIDE_RT_STATS);
```

within. As suggested by the code – profiling is activated on a per pipeline basis.

There are 4 configurations provided with this example. They are designed to be analysed a 2 groups of 2 configurations, where the 2 entries in each group show some development as the configuration changes form one to the other. Therefore configurations A and B are a pair, leaving C and D as a further pair.

5.4.1 Running the application

Within `leon/perfTestbedCfg.h` set the configuration chosen by enabling either A, B, C or D by leaving the chosen option as the only one defined in this file.

Configuration	Guide
A	Single pipeline, running in async mode with one line per iteration
B	Single pipeline, running in async mode with sixteen lines per iteration
C	Dual pipelines (not competing on resource), running in async mode with one line per iteration
D	Dual pipelines (not competing on resource), running in async mode with sixteen lines per iteration

Table 5: Guide to sippPerfTestBed configurations

These configurations are chosen to illustrate the usefulness of running more then one line per iteration in improving SW efficiency (configurations A and B) and also in enabling larger inactive p[periods for the SIPP runtime so that other aspects of the SIPP framework may execute (configurations B and C).

Where appropriate some other aspects which may be of interest will be highlighted.

5.4.2 SW filter efficiency demonstration

Run the test with configuration A – since SIPP_RT_STATS_PRINT_CTRL is set to 1 in mdk/examples/Sipp/SippFw/ma2x5x/sippPerfTestBed/leon/perfTestbedCfg_A.h significant trace will be dumped to the console on completion of the test. The majority of the trace illustrates the timings collected by the runtime profiling for the last frame executed. A sample of the first 20 lines is shown below:

```
UART: Pipeline 0 - Async mode
UART:  Iter    ItTime       HW         SW        CDMA        FW
UART: ======================================================
UART:     0      3488,         0,         0,      2560,      1927
UART:     1      2917,         0,         0,      2239,      1818
UART:     2      2788,         0,         0,      2104,      1558
UART:     3      3181,      2079,         0,      2647,      1652
UART:     4      3120,      2018,         0,      2586,      1632
UART:     5      3255,      2160,         0,      2721,      1765
UART:     6      3400,      2287,      3117,      2835,      1985
UART:     7      3496,      2394,      3213,      2949,      2000
UART:     8      3391,      2289,      3108,      2844,      1979
UART:     9      3509,      2407,      3226,      2962,      2105
```

```
UART:    10      3520,      2418,      3237,      2973,      2023
UART:    11      3418,      2316,      3135,      2871,      2014
UART:    12      3479,      2370,      3196,      2925,      1968
UART:    13      3522,      2420,      3239,      2975,      2019
UART:    14      3388,      2286,      3105,      2841,      1984
UART:    15      3491,      2389,      3208,      2944,      1994
UART:    16      3374,      2272,      3091,      2827,      1970
UART:    17      3617,      2515,      3334,      3070,      2213
UART:    18      3512,      2410,      3229,      2965,      2008
UART:    19      3435,      2333,      3152,      2888,      2031
```

In the case of an async pipeline – the figures relate to clock cycles. The information includes an iteration index (which iteration of the schedule), the total clock cycles for that iteration and then the individual times for the 4 functional units (HW filters / SW filters / CMX DMA and SIPP framework) to perform their required operations on that iteration.

It may be seen, even from the brief sample above that for this pipeline in this configuration when SW filters are required within an iteration, they tend to require the longest time.

This leads the long-pole identification algorithm to state at the end of this trace the following:

```
UART:  ** SW most often long-pole **
```

So in this configuration the profiling and long-pole identification algorithm have said that SW is the bottleneck. That is to say, most gains will be made by analyzing and attempting to optimize software kernel performance. It is helpful to consider the SW performance now.

This test contains only one SW kernel (Box Filter 5x5). There are twelve shaves assigned to it – the maximum available on ma2150 so intuitively performance should be close to optimal? However, it is not so simple. The line width for the filter is 1280 and this is split among the 12 SHAVES so each only processes a small segment of the line - approximately 100 pixels. With short line widths the SHAVE runtime overhead tends to form a higher percentage of the overall total. Each SHAVE suffers the same overhead, though of-course they are *suffered* in parallel. Generally as the line width drops the ratio of useful work cycles to total cycles used on the SHAVE drops. There are two ways to mitigate this. The first is simply to use fewer SHAVES. However, this would result in the efficiency increasing, but the overall SW time required increasing.

The second is to use multiple lines for each iteration. As the guidance provided in 5.3.2.3 suggested, using multiple lines per iteration should always have a strong chance to show some improvement in SW time. The reason is that the efficiency increases and the time involved in the SW kernels increases in rough proportion to the increase in the number of lines per iteration. However the SHAVE runtime overhead increases by a much much smaller amount. Therefore efficiency improves.

To illustrate this point, switch the test to run with configuration B. This configuration asks each filter (HW / SW or CMX DMA) to perform 16 lines of operation on every iteration. The results for the first 20 iterations are as follows:

```
UART: Pipeline 0 - Async mode
UART:  Iter    ItTime       HW         SW        CDMA        FW
UART: ========================================================
UART:     0      8176,         0,         0,      7315,      4215
UART:     1      8155,         0,         0,      7389,      3999
UART:     2     21918,     21204,         0,      7049,      4416
UART:     3     21798,     21084,         0,      7194,      5092
UART:     4     21816,     21088,      9041,      8599,      6903
```

```
UART:     5    21825,     21088,      8819,     11213,      7379
UART:     6    21824,     21087,      8575,     11142,      7212
UART:     7    21837,     21100,      9024,     11544,      7384
UART:     8    21818,     21094,      8548,     11128,      7164
UART:     9    21906,     21182,      8731,     11405,      7388
UART:    10    21832,     21097,      8954,     11330,      7169
UART:    11    21816,     21092,      8720,     11280,      7341
UART:    12    21834,     21099,      8885,     11273,      7132
UART:    13    21817,     21093,      8641,     11433,      7298
UART:    14    21823,     21099,      8492,     10914,      7125
UART:    15    21820,     21089,      8856,     11267,      7324
UART:    16    21812,     21088,      8511,     11234,      7115
UART:    17    21917,     21182,      8761,     11328,      7389
UART:    18    21823,     21092,      8779,     11151,      7129
UART:    19    21824,     21093,      8717,     11168,      7346
```

Of-course the individual iteration figures have increased but as much as 16 times of the output per iteration is being achieved. Looking specifically for now at the SW results. Each iteration time for the SW filter increases only by approximately a factor of 3, despite the fact that 16 times the work is being done! Such a dramatic improvement is a function of the fact that each SHAVE now does about 1600 pixels per iteration on the SW kernels as oppose to about 100 before.

The results should now indicate that the HW has become the long pole in the tests.

Note also that the end results of this include:

- Total test time has dropped – It is now roughly 1/3 of what it was with configuration A
- HW is now the long pole on the pipeline – the efficiency improvements for HW when running with multiple lines per iteration over a single line are much smaller due to the inherently efficient HW operation under all conditions.

5.4.3 Concurrent pipeline scheduling issue demonstration

Run the test with configuration C. This is a dual pipline test runing one line per iteration. In this scenario the key point of interest is not on the clock cycles. Rather the interest focuses on the order in which events happen with configurations C and D.

For interpretation of this information refer to the output log file - mdk/examples/Sipp/SippFw/ma2x5x/sippPerfTestBed/build/myriad2/perfTestBedLog.txt.

This contains the following information after running with configuration C.

```
Test Log : Total test cycles 84592509 LPI 1
<   Time   >|<   Str 0  >|<   Str 1  >|
------------|------------|------------|
      90595 |    F    0 S |            |
    2635632 |    F    0 C |            |
    2728903 |             |    F    0 S |
    5284035 |             |    F    0 C |
    5377790 |    F    1 S |            |
    7922274 |    F    1 C |            |
    8015575 |             |    F    1 S |
   10570564 |             |    F    1 C |
   10664332 |    F    2 S |            |
   13209008 |    F    2 C |            |
```

```
13302303 |              |    F    2 S |
15857328 |              |    F    2 C |
15951078 |    F    3 S |              |
18495728 |    F    3 C |              |
18589018 |              |    F    3 S |
21144146 |              |    F    3 C |
21237908 |    F    4 S |              |
23782922 |    F    4 C |              |
23876214 |              |    F    4 S |
26431364 |              |    F    4 C |
26525124 |    F    5 S |              |
29069648 |    F    5 C |              |
29162931 |              |    F    5 S |
31717735 |              |    F    5 C |
31811512 |    F    6 S |              |
34356900 |    F    6 C |              |
34450191 |              |    F    6 S |
37005096 |              |    F    6 C |
37098858 |    F    7 S |              |
39643714 |    F    7 C |              |
39737001 |              |    F    7 S |
42291811 |              |    F    7 C |
42385582 |    F    8 S |              |
44931092 |    F    8 C |              |
45024382 |              |    F    8 S |
47579130 |              |    F    8 C |
47672880 |    F    9 S |              |
50218094 |    F    9 C |              |
50311396 |              |    F    9 S |
52865983 |              |    F    9 C |
52959752 |    F   10 S |              |
55505224 |    F   10 C |              |
55598519 |              |    F   10 S |
58153601 |              |    F   10 C |
58247352 |    F   11 S |              |
60792158 |    F   11 C |              |
60885439 |              |    F   11 S |
63440195 |              |    F   11 C |
63533958 |    F   12 S |              |
66079424 |    F   12 C |              |
66172713 |              |    F   12 S |
68727644 |              |    F   12 C |
68821400 |    F   13 S |              |
71366190 |    F   13 C |              |
71459471 |              |    F   13 S |
74014312 |              |    F   13 C |
74108086 |    F   14 S |              |
76652932 |    F   14 C |              |
76746219 |              |    F   14 S |
79301317 |              |    F   14 C |
79395066 |    F   15 S |              |
81939719 |    F   15 C |              |
82033005 |              |    F   15 S |
84588179 |              |    F   15 C |
```

This illustrates the occurrence of events within the application. Each line signifies a new event. “F X S” is frame X starting for the pipeline within whose column the message is located. “F X C” is frame X completing.

It may be observed that with configuration C the pipelines are effectively running serially. Once any pipeline

starts a frame, the subsequent event will be the completion of the frame on that pipeline. Never are the two pipelines running in parallel. This is due to the execution experience detailed in section 5.3.3. Once the framework runtime has completed its operations for the subsequent iteration, the HW, SW and CMX DMA interrupts are lined up in sequence meaning that the processor does not return form interrupt context for a sufficiently long period to enable the access scheduler to post the other pipeline's process frame request onto the runtime.

Configuration D runs the same test, but this time with 16 lines per iteration. This time the output logfile can be expected to contain something like the following

```
Test Log : Total test cycles 20325024 LPI 16
<   Time   >|<   Str 0  >|<   Str 1  >|
------------|------------|------------|
      91251 |   F    0 S |            |
     257972 |      V     |   F    0 S |
    1148463 |   F    0 C |      V     |
    1312810 |   F    1 S |      V     |
    1350661 |      V     |   F    0 C |
    1526899 |      V     |   F    1 S |
    2379197 |   F    1 C |      V     |
    2553105 |   F    2 S |      V     |
    2604046 |      V     |   F    1 C |
    2783351 |      V     |   F    2 S |
    3618095 |   F    2 C |      V     |
    3793842 |   F    3 S |      V     |
    3860614 |      V     |   F    2 C |
    4038300 |      V     |   F    3 S |
    4860156 |   F    3 C |      V     |
    5037649 |   F    4 S |      V     |
    5126651 |      V     |   F    3 C |
    5304938 |      V     |   F    4 S |
    6102450 |   F    4 C |      V     |
    6279877 |   F    5 S |      V     |
    6392641 |      V     |   F    4 C |
    6570558 |      V     |   F    5 S |
    7347716 |   F    5 C |      V     |
    7525617 |   F    6 S |      V     |
    7659311 |      V     |   F    5 C |
    7837410 |      V     |   F    6 S |
    8593023 |   F    6 C |      V     |
    8770638 |   F    7 S |      V     |
    8926332 |      V     |   F    6 C |
    9103918 |      V     |   F    7 S |
    9837406 |   F    7 C |      V     |
   10015101 |   F    8 S |      V     |
   10193945 |      V     |   F    7 C |
   10371274 |      V     |   F    8 S |
   11082970 |   F    8 C |      V     |
   11260563 |   F    9 S |      V     |
   11463087 |      V     |   F    8 C |
   11641326 |      V     |   F    9 S |
   12329820 |   F    9 C |      V     |
   12507416 |   F   10 S |      V     |
   12732020 |      V     |   F    9 C |
   12910018 |      V     |   F   10 S |
   13576939 |   F   10 C |      V     |
   13754687 |   F   11 S |      V     |
   13998502 |      V     |   F   10 C |
   14176174 |      V     |   F   11 S |
```

```
14821433 |   F   11 C |       V     |
14999053 |   F   12 S |       V     |
15265492 |       V     |   F   11 C |
15443400 |       V     |   F   12 S |
16066558 |   F   12 C |       V     |
16244364 |   F   13 S |       V     |
16532303 |       V     |   F   12 C |
16709840 |       V     |   F   13 S |
17310247 |   F   13 C |       V     |
17488081 |   F   14 S |       V     |
17797912 |       V     |   F   13 C |
17975594 |       V     |   F   14 S |
18555441 |   F   14 C |       V     |
18733108 |   F   15 S |       V     |
19065144 |       V     |   F   14 C |
19243240 |       V     |   F   15 S |
19797486 |   F   15 C |       V     |
20322036 |             |   F   15 C |
```

In this case it may be seen that while events are occurring on one pipeline, the alternate pipeline is often in progress (as indicated by a “V” in the log). The start and complete events form the two pipes are now interspersed in the manner to be expected of true concurrent operation.

Note also the total test time has reduced form 85 M cycles to approx 20 M cycles. Some of this is due to the efficiency increase in the SW filters in using 16 line per iteration but of-course the fact that the HW and SHAVE resource is now truly running in parallel is also a factor.

Perhaps an equally interesting result can be obtained by switching back to configuration C but modifying mdk/examples/Sipp/SippFw/ma2x5x/sippPerfTestBed/leon/perfTestbedPipes.c so that each pipeline is allocated only one SHAVE (as opposed to 6 each). The modification required is to search for the lines:

```
#elif PERF_TESTBED_NUM_PIPELINES > 0x1
u32                 sliceFirst[PERF_TESTBED_NUM_PIPELINES] = {0,6};
u32                 sliceLast[PERF_TESTBED_NUM_PIPELINES]  = {5,11};
#else
```

and change to

```
#elif PERF_TESTBED_NUM_PIPELINES > 0x1
u32                 sliceFirst[PERF_TESTBED_NUM_PIPELINES] = {0,6};
u32                 sliceLast[PERF_TESTBED_NUM_PIPELINES]  = {0,6};
#else
```

This change of-course slows down the SW kernels per pipeline – however this slowdown simply presents an opportunity for the runtime to service interrupts from the alternate pipeline in the intervening period. This enables the two pipelines to run in parallel as desired. In fact the overall test time only increased from about 85M clock cycles to 89M clock cycles, despite now allocating only 1/6th of the SHAVE resource!

This illustrates that when running more than one pipeline in asynchronous mode, it is often the time taken to service interrupts and the interrupt latency of the system which will be the defining factor in throughput. The addition of additional resource – such as in the example just shown adding 5 more SHAVES per pipeline may not in itself make a significant difference. Running multiple lines per iteration not only increases shave efficiency and therefore helps justify the addition of more SHAVE resource to a pipeline, it also reduces the total number of interrupts taken which can be a crucial factor.

6 MA2150 and MA2100 SIPP comparison

6.1 Target silicon changes

Some background on the target silicon for the respective SIPP frameworks is informative in aiding understanding of some of the differences between them.

MA2150 saw the introduction of an optimized ISP solution employing the SIPP hardware filters called oPipe. In the oPipe the output of each filter in the pipeline is connected directly to the next filter in the pipeline where it fills the local line buffer (LLB) if present or is processed directly (without any copy to/from CMX memory). The introduction of the LLBs greatly reduces memory bandwidth and power consumption in the oPipe and when the filters are used in the context of the SIPP framework. These LLBs do mean that context switching those SIPP HW filters which contain them is no longer possible. The consequence of this is that no sub-frame granularity of operation is available.

In the context of the SIPP framework this means that a HW filter may no longer appear multiple times in the same pipeline on MA2150. It further means that the `sippProcessIters()` API available on MA2100 is no longer available on MA2150 as there is no means to restore the context to the HW filters to enable restarts within a frame.

Further MA2150 has seen the introduction of two other features which have significant effect on the framework. The first is the introduction of filters with multiple output buffers. MA2100 framework could assume that all filters had only one output buffer. On MA2150 this is no longer the case since the Debayer filter is capable of producing a RGB and a Luma output. Next is the concept of an inherent latency of operation. On MA2100 each filter would produce one line of output on each so it could be said it had zero latency. On MA2150, due to the composited nature of certain HW filters, these filters need to be called several times before the first line of output is produced. The MA2150 framework of-course takes care of this to ensure that all subsequent filters in the pipe are correctly scheduled.

6.2 API

When producing the MA2150 API, a strong influencing factor was to maintain compatibility with the existing MA2100 API. The intention is that this will mean existing applications should be capable of working with minimal modification when targeting the new framework, providing of-course that the pipelines created within the applications target filters which continue to be supported. With this in mind the majority of the API functions currently exported by the MA2100 framework will be maintained. The new features detailed in section 6.3 are implemented through the addition of new APIs.

The API is described in full in section 7.

6.3 New MA2150 framework features

6.3.1 Asynchronous API

The addition of an asynchronous API – `sippProcessFrameNB()` provides clients to the SIPP framework with the scope to continue to use the LEON resource while the HW and/or SHAVE filters are operational during the course of a frame.

Servicing of the HW filters and SHAVEs involved in the frame during its operation by the SIPP framework will occur in interrupt context. Naturally this may add some cycles to the overall frame processing time since at

a bare minimum interrupt latency and time to execute interrupt handling framework code will be added per iteration. However, this additional overhead is seen as worthwhile in order to free the processor to perform other latency dependent tasks during the frame. Further the continued provision of the synchronous blocking API supports those situations when maximum performance is critical.

6.3.2 Multi-pipe concurrent interface

Within the framework an access scheduler has been introduced to provide a multi-pipeline to the client while maintaining control on the access of the pending processing instructions to the HW and SHAVE resource each pipe demands. This interface works in association with the asynchronous API discussed in 6.3.1, allowing the client to post `sippProcessFrameNB`() operations for several pipelines in sequence without waiting for that operation to be complete. The framework will control allocation of each pipeline to the underlying resource in turn based on a scheduling algorithm which has the potential to be tuned to a clients needs if prioritization of pipelines is required in some way.

7 MA2x5x SIPP API

7.1 API function calls

7.1.1 sippPlatformInit()

7.1.1.1 Prototype

```
void sippPlatformInit();
```

7.1.1.2 Description

This function initializes the SIPP framework AND also initializes the Myriad system e.g. clocks, hardware accelerators.

If the application require control over initializing the Myriad device then it may be better to use the `sippInitialize()` function.

7.1.1.3 Parameters

–	No parameters.

7.1.2 sippInitialize()

7.1.2.1 Prototype

```
void sippInitialize();
```

7.1.2.2 Description

This function initializes the SIPP internals only. Use this function if the Myriad devices is initialized separately in the application.

7.1.2.3 Parameters

–	No parameters.

7.1.3 sippCreatePipeline()

7.1.3.1 Prototype

```
SippPipeline* sippCreatePipeline(u32 shaveFirst,
                                 u32 shaveLast,
                                 u8 *mbinImg);
```

7.1.3.2 Description

This is the first API function your application should call, in order to instantiate a SIPP pipeline. A pointer referring to the SIPP pipeline is returned, which can be passed to other SIPP API functions.

7.1.3.3 Parameters

shaveFirst, shaveLast	These two parameters specify which SHAVE processors are to be assigned to execute the SIPP pipeline. They specify an inclusive, contiguous and zero-based set of SHAVEs. shaveFirst must be <= shaveLast. For example, if 0 and 3 were specified for shaveFirst and shaveLast respectively, then SHAVEs 0, 1, 2 and 3 would be assigned to the pipeline. It is up to the application to manage SHAVE allocation. You must decide which SHAVE processors to allocate processing by the pipeline being instantiated, and which, if any, to assign to other processing tasks.
mbinImg	As part of the build process for a SIPP application, an mbin (Myriad Binary) is created, which contains all of the code (SIPP framework components and filters) which will run on the SHAVE processors. At runtime, this code gets loaded into the CMX slice associated with each of the SHAVE processors assigned to the pipeline. This parameter points to the mbin image containing the code targeted to run on the SHAVE processor. For maximum forward portability, you should wrap this parameter with the SIPP_MBIN macro.

7.1.3.4 Example

```
pl = sippCreatePipeline(1, 3, SIPP_MBIN(mbinSippImg));
```

7.1.4 sippCreateFilter()

7.1.4.1 Prototype

```
SippFilter* sippCreateFilter(SippPipeline *pl,
                             u32 flags,
                             u32 outW,
                             u32 outH,
                             u32 numPl,
                             u32 bpp,
                             u32 paramsAlloc,
                             void (*funcSvuRun)(struct SippFilterS
                                  *fptr, int svuNo, int runNo),
                             const char *name);
```

7.1.4.2 Description

This function instantiates a SIPP filter and associates it with the specified pipeline, *pl*. A pointer referring to the SIPP filter is returned. When instantiating a given type of filter, refer to the filter-specific documentation to ensure that you pass parameters that are valid for and compatible with that specific type of filter (supported pixel depths, number of planes supported, size of configuration parameters structure, function name of the SHAVE entry point etc).

7.1.4.3 Parameters

Parameter	Description
pl	A reference to a pipeline, returned by *sippCreatePipeline()*. The instantiated filter will be associated with this pipeline.

Parameter	Description
flags	Currently the only defined flag is `SIPP_RESIZE`. This flag should be passed if the filter input resolution is not the same as the output resolution.
outW	Width of the frame to be output by this filter.
outH	Height of the frame to be output by this filter.
numPl	Number of planes of data in the filter's output buffer.
Bpp	Bytes per pixel of the output buffer data.
paramsAlloc	Number of bytes needed to store configuration parameters for the type of filter being instantiated.
funcSvuRun	The filter's main entry point. This is a pointer to a function which runs on the `SHAVE` processor. When invoked, it will produce one scanline of output data (single or multi-plane). When multiple `SHAVE`s are assigned to the pipeline, each `SHAVE` is responsible for outputting a segment of the scanline.
name	For debug purposes only. Character string to identify the filter, which has meaning within the application.

7.1.5 sippLinkFilter()

7.1.5.1 Prototype

```
void sippLinkFilter(SippFilter *fptr,
                    SippFilter *parent,
                    u32 nLinesUsed,
                    u32 hKerSz);
```

7.1.5.2 Description

This function is used to link the filters which have been instantiated within a pipeline into a graph. It establishes a parent/consumer relationship between a pair of filters.

7.1.5.3 Parameters

Parameter	Description
fptr	The child or consumer in the relationship.
parent	The parent or producer in the relationship.
nLinesUsed	Number of lines that the child filter will reference in the parent filter's output buffer. For example, if the child filter performs a 5x5 convolution on the data from the parent filter's output buffer, the number of "used" lines is 5. The framework will look at the requirements of all of the consumers of an output buffer, to automatically determine how many lines of data actually need to be allocated in the output buffer.
hKerSz	Horizontal kernel size (in pixels) for horizontal padding.

7.1.6 sippLinkFilterSetOBuf()

7.1.6.1 Prototype

```
u8 sippLinkFilterSetOBuf (SippFilter * pFilter,
                          SippFilter * pParent,
                          u32          parentOBufIdx)
```

7.1.6.2 Description

This function is used to when a parent filter has more than one output buffer. By default `sippLinkFilter` links the consumer filter to output buffer 0 of the parent filter (info on what this is should be published for all such filters). This function may be used in conjunction with `sippLinkFilter` to switch the link to any of the other parent output buffers available. Note that if a child wished to consume form more than one parent buffer then two such links should be made, and those links adjusted where necessary via this API.

7.1.6.3 Parameters

Parameter	Description
pFilter	The child or consumer in the relationship.
pParent	The parent or producer in the relationship.
parentOBufIdx	The parent output buffer index that the child will consume from.

7.1.7 sippFinalizePipeline()

7.1.7.1 Prototype

```
void sippFinalizePipeline(SippPipeline *pl);
```

7.1.7.2 Description

This function is used to compute the frame schedule and prepares the pipeline(s) for execution. If it is not called the schedule is initialized the first time the *sippProcessFrame* or sippAllocCmxMemRegion is called.

By default, the SIPP uses the CMX slices allocated to the SIPP pipeline (via `sippCreatePipeline`) as a temporary workspace to calculate the schedule. For large pipelines this may not be enough so it is possible to re-use other application memory by calling the function, `sippInitSchedPoolArb`, and pointing at application memory e.g. Frame buffer area.

7.1.7.3 Parameters

Parameter	Description
pl	Pointer reference to the pipeline.

7.1.8 sippProcessFrame()

7.1.8.1 Prototype

```
void sippProcessFrame(SippPipeline *pl);
```

7.1.8.2 Description

Invokes the SIPP scheduler to process 1 frame-worth of data. A single frame of data will be output by the source filters, and the data will be passed from filter to filter, until a full frame of data has been output by the sink filters.

7.1.8.3 Parameters

Parameter	Description
pl	Pointer reference to the pipeline to run.

7.1.8.4 Notes

This is a blocking API and should only be called when no other API calls are outstanding (via the async API interface).

7.1.9 sippProcessFrameNB()

7.1.9.1 Prototype

```
void sippProcessFrameNB(SippPipeline *pl);
```

7.1.9.2 Description

Invokes the SIPP scheduler to process 1 frame-worth of data. A single frame of data will be output by the source filters, and the data will be passed from filter to filter, until a full frame of data has been output by the sink filters. Notification of successful completion will be returned to the client via the callback function the client registered with sippRegisterEventCallback.

7.1.9.3 Parameters

Parameter	Description
pl	Pointer reference to the pipeline to run.

7.1.9.4 Notes

This is a non-blocking API variant of sippProcessFrame.

7.1.10 sippReschedule()

7.1.10.1 Prototype

```
void sippReschedule(SippPipeline *pl);
```

7.1.10.2 Description

This function is called if the application needs to re-calculate the pipeline execution schedule.

The pipeline schedule is normally calculated once at application initialization. Typical situations where the schedule needs to be recalculated is if the image or frame resolution changes or if the application needs to change a parameter e.g. kernel size in one or more of the kernels in the pipeline.

If the application needs to reschedule pipelines then it must specify a buffer area to store the schedule (by default in applications that don't re-schedule the SIPP handles the schedule buffer area). The function to set the schedule buffer area is `sippInitSchedPoolArb()`.

7.1.10.3 Parameters

Parameter	Description
pl	Pointer reference to the pipeline to run.

7.1.11 sippAllocCmxMemRegion()

7.1.11.1 Prototype

```
Int32 sippAllocCmxMemRegion (SippPipeline * pipe, SippMemRegion *
memRegList);
```

7.1.11.2 Description

Provides an additional means to allocate memory to a SIPP framework pipeline for use in the allocation of HW SIPP filter line buffers, excluding DMA filters. The chief intention of this API is to provide a memory efficient mechanism to ameliorate the bandwidth efficiency of pipelines containing multiple HW filters operating in parallel by spreading the line buffer areas over multiple CMX slices. This mechanism will reduce the memory access contention of the HW filters.

7.1.11.3 Parameters

Parameter	Description
pipe	Previously created SippPipeline struct to which the memory regions passed are to be assigned.
memRegList	Pointer to NULL terminated list of `SippMemRegion` structures.

7.1.11.4 Usage Notes

- The `sippCreatePipeline()` API has 2 parameters `sliceFirst` and `sliceLast`. All such CMX slices assigned at pipline creation are still assumed available to the pipeline and so not need not appear in this additional list.
- Line buffers are always aligned to 8 byte boundaries – therefore ideally the `regionOffset` members of the `SippMemRegion` structs will be 8-byte aligned. To be otherwise is not an error, but the SIPP framework will internally consider the start location to be the first 8 byte aligned address falling within the region.
- Reaffirmation that the `memRegList` parameter to the API must be a NULL-terminated list of `SippMemRegion` structures. For example a client could statically allocate the following in c-code:

```
SippMemRegion CmxMemRegions[] =
{
  {
    .regionOffset = 0x0,
    .regionSize = 0x1000,
    .regionUsed = 0x0,
  },
  {
    .regionOffset = 0x8000,
    .regionSize = 0x1000,
    .regionUsed = 0x0,
  },
  { 0, 0, 0 } /* Null terminated List */
};
```

- `CmxMemRegions` is then a suitable parameter to the API. Usage of this API is chiefly recommended for pipelines which involve only SIPP HW filters. For pipelines employing a mix of SW / HW filters, the chunking of the image will continue to be dictated by the number of SHAVES allocated to the pipeline. Memory regions allocated via this API which are located in CMX memory slices not linked to one of the allocated SHAVES will be used for the output line buffers of those HW filters having exclusively HW consumer filters. The framework will check that the regions provided are suitable for accepting chunks of such a HW filter's output by considering the size available and that areas may be found which are spaced at the requisite distance apart (almost certainly CMX slice size) to enable consistency with the pipeline chunking.
- If only one SHAVE is employed, then chunking is not used (or more accurately chunk size is equal to line width). In this case the slice stride is not relevant and so the restrictions on the memory regions are reduced.
- Should the client desire to setup memory regions which conform to a slice stride not equal to the default value (i.e. the CMX region size of 128 Kb) then a call to void `sippSetSliceSize(UInt32 size)` should be made to establish the new slice size before the call to `sippCreatePipeline()` for the pipeline – the framework will NOT attempt to find some suitable slice stride which could be fitted to the memory regions provided as the chances of being able to find a suitable quantity in a randomly allocated set of memory regions are remote. Therefore the client should consider this stride as part of its memory allocation process.
- Individual HW filter output line buffers (for individual planes) must be in a single section – that is to say the full line buffer must be in contiguous memory. So the regions allocated must permit allocation of these contiguous regions. If chunking is used the line buffers are chunked, but the separate lines (for each plane) within each chunk are in contiguous memory).
- In order to avoid contention among the various HW filters in use an ideal assignment of memory regions

would permit the framework to distribute the line buffers for the HW filters in use among a wide spread of CMX slices

- The API should be called before any other call to `sippFinalizePipeline()` as this mechanism needs to be registered before the triggers of memory allocation within `sippFinalizePipeline` are called. In turn this API will itself call `sippFinalizePipeline()`

7.1.11.5 Constraints

- Regions should not straddle a CMX slice boundary
- Regions must facilitate the allocation of line buffers in such a way that a uniform slice stride value may be used if chunking of the lines is applied (note chunking need not be applied in a pipeline consisting of only HW filters, or of HW filters and only one SHAVE performing the SW filters). In order to constitute a chain of regions suitable for chunking, the region start locations must be spaced at a distance exactly equal to the slice stride. It is recommended that the default slice stride of 128 Kb is maintained where possible as this leads to optimal performance in mixed HW / SW pipelines. This means that when additional regions are allocated via this API, they should be spaced at 128 Kb distance, or more appropriately they should be at the same offset form the start of the CMX slice in which they are contained. If the slice stride is not set to 128 Kb, HW constraints on the slice stride must be adhered to. A single global slice size is programmable. The minimum size is 32 Kb and the maximum is 480 Kb, programmable in increments of 32 Kb.

7.1.12 SippChooseMemPool

7.1.12.1 Prototype

```
void   sippChooseMemPool (ptSippMCB       pSippMCB,
                          SippVirtualPool vPool,
                          u32             physPoolIdx);
```

7.1.12.2 Description

This function is called to remap a virtual pool controlled by a certain SIPP Memory Control Block (accessed through a pointer of type pSippMCB) to be mapped to a physical pool – either the CMX pool or the DDR pool.

7.1.12.3 Parameters

Parameter	Description
pSippMCB	Pointer reference to a memory control block.
vPool	Virtual Pool to remap.
physPoolIdx	Set to 0 to remap to CMX pool or 1 to remap to DDR pool.

7.1.13 sippRegisterEventCallback()

7.1.13.1 Prototype

```
void sippRegisterEventCallback (SippPipeline *      pPipe,
                                sippEventCallback_t pfCallback );
```

7.1.13.2 Description

This function is called when the client wishes to register a callback handler to receive events pertaining to the specified pipeline. The pPipe parameter will be returned to the callback to enable the client to distinguish the source of the callback.

7.1.13.3 Parameters

Parameter	Description
pPipe	Pointer reference to the pipeline.
pfCallback	Pointer to a callback function to be executed by the framework when an event occurs for the specified pipeline.

7.1.14 sippDeletePipeline()

7.1.14.1 Prototype

```
void sippDeletePipeline (SippPipeline * pPipe);
```

7.1.14.2 Description

Informs the framework that a specified pipeline is no longer to be used and so its internal resources may be deleted. This includes all memory regions

7.1.14.3 Parameters

Parameter	Description
pPipe	Pointer reference to the pipeline.

7.1.15 sippFilterAddOBuf()

7.1.15.1 Prototype

```
void sippFilterAddOBuf(pSippFilter pFilter,
                       u32         numPlanes,
                       u32         bpp);
```

7.1.15.2 Description

This function is called to add a new output buffer to filters capable of producing more than one output. It is assumed the vertical and horizontal dimensions of all output buffers are constant for a filter so the output buffer inherits the dimensions established for the filter during its creation via sippCreateFilter. However there is scope for the additional output buffer to have a differing number of planes and output format and so these two variables are established via input parameters.

7.1.15.3 Parameters

Parameter	Description
pFilter	Pointer reference to the filter.
numPlanes	Number of planes for the new output buffer.
bpp	Bytes per pixel of the new output buffer's data.

7.1.16 sippFilterSetBufBitsPP

7.1.16.1 Prototype

```
void sippFilterSetBufBitsPP (pSippFilter pFilter,
                             u32         oBufIdx,
                             u32         bitsPerPixel);
```

7.1.16.2 Description

This function is called to allow an output output buffer type to be modified for a filter. This API allows the size to be expressed in terms of bits, thus allowing support for packed formats.

7.1.16.3 Parameters

Parameter	Description
pFilter	Pointer reference to the filter.
oBufIdx	Identifies which of the filter's output buffers to modify.
bpp	bits per pixel of the new output buffer's data.

7.1.17 sippPipeSetLinesPerIter

7.1.17.1 Prototype

```
void sippPipeSetLinesPerIter (pSippPipeline pPipe,
                              u32           linesPerIter);
```

7.1.17.2 Description

Optional – by default the SIPP uses one line per iteration of any created schedule. This API allows the client to modify this to 2/4/8 or 16. This has performance benefits in certain circumstances while increasing the line buffer memory requirement.

7.1.17.3 Parameters

Parameter	Description
pPipe	Pointer reference to the pipeline.
linesPerIter	Number of lines to run per iteration for each scheduled filter.

7.1.18 sippInitSchedPoolArb()

7.1.18.1 Prototype

```
void sippInitSchedPoolArb(u8 *addr, u32 size);
```

7.1.18.2 Description

Optional – by default, the SIPP uses the CMX slices allocated to the SIPP pipeline (via sippCreatePipeline) as a temporary workspace to calculate the schedule.

This function allows the user specify an alternative memory area for the SIPP to use to calculate the runtime schedule.

Typically the caller should define a buffer somewhere in DDR or CMX and point SIPP to that. The application can re-use this memory e.g. frame buffer memory.

NOTE: This function must be used if the pipeline application reschedules the pipeline.

7.1.18.3 Parameters

Parameter	Description
addr	Buffer address for (temporary) schedule calculation work area.
size	Size of buffer.

7.1.19 sippRdFileU8()

7.1.19.1 Prototype

```
void sippRdFileU8(u8 *buff, int count, const char *fName);
```

7.1.19.2 Description

Read a file containing unsigned 8bit integers.

7.1.19.3 Parameters

Parameter	Description
buff	Buffer address for data read from file.
count	Number of bytes.
fName	File name.

7.1.20 sippWrFileU8()

7.1.20.1 Prototype

```
void sippWrFileU8(u8 *buff, int count, const char *fName);
```

7.1.20.2 Description

Write unsigned 8bit integers to a file.

7.1.20.3 Parameters

Parameter	Description
buff	Buffer address with data to write to file.
count	Number of bytes.
fName	File name.

7.1.21 sippRdFileU8toF16()

7.1.21.1 Prototype

```
void sippRdFileU8toF16(u8 *buff, int count, const char *fName);
```

7.1.21.2 Description

Read a file of 8-bit integers and convert and store as 16-bit Floats (Half-float).

7.1.21.3 Parameters

Parameter	Description
buff	Buffer address for data read from file.
count	Number of bytes.
fName	File name.

7.1.22 sippWrFileF16toU8

7.1.22.1 Prototype

```
void sippWrFileF16toU8(u8 *buff, int count, const char *fName);
```

7.1.22.2 Description

Compare two unsigned 8-bit integer (char) buffers.

7.1.22.3 Parameters

Parameter	Description
refA	First buffer.
refB	2nd buffer for comparison.
len	Number of bytes to compare.

7.1.23 sippDbgCompareU16

7.1.23.1 Prototype

```
void sippDbgCompareU16(u16 *refA, u16 *refB, int len);
```

7.1.23.2 Description

Compare two unsigned 16-bit buffers values.

7.1.23.3 Parameters

Parameter	Description
refA	First buffer.
refB	2nd buffer for comparison.
len	Number of bytes to compare.

7.1.24 sippDbgCompareU32

7.1.24.1 Prototype

```
void sippDbgCompareU32(u32 *refA, u32 *refB, int len);
```

7.1.24.2 Description

Compare two unsigned 32-bit unsigned integer buffers.

7.1.24.3 Parameters

Parameter	Description
refA	First buffer.
refB	2nd buffer for comparison.
len	Number of bytes to compare.

7.1.25 sippErrorSetFatal

7.1.25.1 Prototype

```
void sippErrorSetFatal (u32 errCode)
```

7.1.25.2 Description

Sets the error provided in the parameter as a fatal error which will trigger an abort to allow immediate debug.

7.1.25.3 Parameters

Parameter	Description
errCode	A single error code to be marked as fatal if occurs.

7.1.26 sippGetLastError

7.1.26.1 Prototype

```
u32 sippGetLastError ( )
```

7.1.26.2 Description

Returns last error recorded by the framework

7.1.27 sippGetErrorHistory

7.1.27.1 Prototype

```
u32 sippGetErrorHistory (u32 * ptrErrList)
```

7.1.27.2 Description

Returns the last errors recorded up to a maximum of the cfg define SIPP_ERROR_HISTORY_SIZE. This is numbered from boot or the last call to this function.

7.1.27.3 Parameters

Parameter	Description
ptrErrList	A pointer to client allocated storage error to hold the error list. Area needs to be sized at sizeof(u32) * SIPP_ERROR_HISTORY_SIZE where this macro is as defined in sippCfg.h

7.1.28 sippPipeGetErrorStatus

7.1.28.1 Prototype

```
u32  sippPipeGetErrorStatus (SippPipeline * pPipe)
```

7.1.28.2 Description

Returns 0x1 if any error code has been recorded on the specified pipeline

7.1.28.3 Parameters

Parameter	Description
pPipe	Pointer reference to the pipeline.

7.1.29 sippPipeSetNumLinesPerBuf

7.1.29.1 Prototype

```
void sippPipeSetNumLinesPerBuf (pSippFilter pFilter,
```

```
                                u32          oBufIdx,
                                u32          numLines)
```

7.1.29.2 Description

Allows the client to manually set the number of buffer lines to be used for any pipeline buffer when the direct streaming runtime (oPipe runtime) has been chosen by the framework. This enables the client to consider reduction of the buffer size allocated by the algorithm if empirical testing shows any performance degradation is considered justifiable in order to gain a decreased memory footprint.

7.1.29.3 Parameters

Parameter	Description
pFilter	Pointer reference to the filter.
oBufIdx	Index into the filter's list of output buffers.
numLines	The number of lines to size the specified buffer to.

7.2 Callback event list

Table 6 below describes the events which may be produced by a pipeline when running asynchronously in response to a call to `sippProcessFrameNB`.

Event	Description
eSIPP_PIPELINE_FINALISED	Pipeline has been finalized.
eSIPP_PIPELINE_RESCHEDULED	Pipeline rescheduling is complete.
eSIPP_PIPELINE_FRAME_DONE	A pipeline has sent one frame's worth of data to output sink.
eSIPP_PIPELINE_ITERS_DONE	*No longer supported.*
eSIPP_PIPELINE_SYNC_OP_DONE	*Internal event only.*
eSIPP_PIPELINE_STARTED	Pipeline is internally scheduled to commence execution.

Table 6: Callback events table

NOTE: Callbacks may be made from any context (thread / IRQ) so the client callback function should be aware of this.

7.3 Pipeline Flags

Each created pipeline contains a flag member (SippPipeline::u32 flags). The SIPP framework client may choose to set certain bits within this 32-bit variable to trigger specified behavior within the SIPP framework.

A guide to the flag to bit mapping and what they achieve when set is detailed in Table 7. Not all flags are designed g set by the client

Flag	Value	Client Access	Description
PLF_REQUIRES_SW_PADDING	(1<<0)	R	Allows Client to understand if SHAVE padding has been required in pipe.
PLF_UNIQUE_SVU_CODE_SECT	(1<<1)	N/A	To be removed.
PLF_IS_FINALIZED	(1<<2)	R	Set when pipeline schedule is created.
PLF_MAP_SVU_CODE_IN_DDR	(1<<3)	W	Set to force the shave code to be placed in DDR – data remains in CMX.
PLF_RUNS_ITER_GROUPS	(1<<4)	W	Set to create CMX DMA descriptors to allow SHAVE data space to be saved to DDR.
PLF_DISABLE_OPIPE_CONS	(1<<5)	W	Force the framework to ignore possible oPipe connections in a pipeline when they are available – useful if desire to attach.
PLF_PROVIDE_RT_STATS	(1<<6)	W	Runtime statistics are collected (see section XX for details).
PLF_ENABLE_SW_QU_USE	(1<<7)	W	Enable SW command queue usage, potentially increasing performance.
PLF_CONSIDER_OPIPE_RT	(1<<8)	W	Consider the oPipe runtime as an option for the pipe.

Table 7: Pipeline flags table

7.4 Error management and reporting

7.4.1 Error flags

Table 8 details the error flags now produced by the framework.

Code	Description
E_OUT_OF_MEM	Out of memory (a requirement on a memory pool could not be satisfied). Try increasing pool size.
E_INVALID_MEM_P	Invalid Memory Pool.
E_PAR_NOT_FOUND	Parent not found (a filter is looking for a parent and is not found in parent list). Internal Error.
E_DATA_NOT_FOUND	Internal algorithm failure. Internal Error.
E_RUN_DON_T_KNOW	Scheduler: Cannot schedule filter. Pipeline or internal error

Code	Description
E_INVALID_HW_PARAM	Invalid HW Parameter.
E_INVLD_FILT_FIRST_SLICE	First slice of a filter is smaller than first slice of its pipeline. Pipeline creation failure.
E_INVLD_FILT_LAST_SLICE	Last slice of a filter is larger than last slice of its pipeline. Pipeline creation failure.
E_MISSING_SHAVE_IMAGE	If pipeline uses SW filters, but shave image is NULL. Pipeline creation failure.
E_UNIMPLEMENTED_FEAT	Marks unimplemented feature.
E_PC_CMX_MEM_ALLOC_ERR	On PC builds only: marks that CMX memory buffer could not be allocated.
E_OPT_EXEC_NUM	
E_CANNOT_FINISH_FILTER	Scheduler cannot finish schedule of a filter till filter height is reached. Can happen for very small image heights. Internal error.
E_DATA_ALIGN	Unused on ma2x5x silicon.
E_INVLD_MIPI_RX_LOOPBACK	Loopback not supported for the given mipi RX units (only RX 1 and 3 support loopback).
E_TOO_MANY_FILTERS	Maximum permitted number of filters was exceeded, cannot be > SIPP_MAX_FILTERS_PER_PIPELINE as set in sippCfg.h
E_INVLD_MULTI_INSTANCE	Invalid use of multi-instance: applies to all HW filters!
E_INVLD_HW_ID	Unused with ma2x5x silicon.
E_TOO_MANY_PARENTS	A filter's number of parents exceeds SIPP_FILTER_MAX_PARENTS as defined in sippCfg.h
E_TOO_MANY_CONSUMERS	A filter's number of consumers exceeds SIPP_FILTER_MAX_CONSUMERS as defined in sippCfg.h
E_RUNS_ITER_GROUPS	Unused with ms2x5x SIPP.
E_TOO_MANY_DMAS	Number of DMA filters defined in the pipeline exceeds SIPP_MAX_DMA_FILTERS_PER_PIPELINE as defined in sippCfg.h
E_INVLD_SLICE_WIDTH	Invalid Slice width (must be multiple of 8) for a SW filter. Attention should be paid when using resizing SW kernels to ensure output slice width remains a multiple of 8.
E_OSE_CREATION_ERROR	An issue arose from an attempt to create OSEs in pipeline. Internal Error.
E_CDMA_QU_OVERFLOW	CMX DMA task qu has overflowed.
E_PC_RUNTIME_FAILURE	PC Model experienced runtime error.
E_SCHEDULING_OVF	Too many events queued in scheduler. Modify app to wait on event completion or increase scheduler event queue size via SIPP_ACCESS_SCHEDULER_QU_SIZE in sippAccessSchedulerTypes.h
E_BLOCK_CALL_REJECTED	A blocking API call was rejected as pending operations remain

Code	Description
E_PRECOMP_SCHED	An error was detected in an attempt to use a precompiled schedule for the pipeline.
E_FINALISE_FAIL	An attempt to finalise a pipe failed often due to one of the errors above but this allows a catch all test.
E_HEAP_CREATION_FAIL	An error occurred in attempt to create a heap.

Table 8: Error code descriptions

7.4.2 Error management API best practice

Some of the errors listed in Table 8 have a pipeline scope, some a filter scope and some a framework scope. In order to guide the client with further information a suite of error management APIs has been added to enable a specific scope to be interrogated for the occurrence of errors in previously enabled APIs.

`SippGetLastError()` (see section 7.1.26) allows a check to see the last error recorded at any level within the framework while `sippGetErrorHistory()` (see section 7.1.27) returns all recorded errors since boot or the last call to the function. For a more pipeline specific scope, `sippPipeGetErrorStatus()` (see section 7.1.28) may be used to check out if nay errors have occurred which are specific to that pipeline. This is a useful tool to call after an API call to check on the pipeline status. If this API call indicates an error has occurred, the previously mentioned APIs may then be used to interrogate the framework for more information on the error type.

Further some tuning of the response to errors is facilitated. `SippErrorSetFatal()` (see section 7.1.25) allows any error to be marked as fatal so that the framework will trap the processor on its occurrence. This can facilitate more efficient debug under certain circumstances.

NOTE: It is planned that in MDK releases subsequent to 16.06 SIPP examples will be re-tooled to use the error management APIs described here.

8 SIPP Hardware accelerator filters

Below is a summary of hardware filters, along with supported input and output formats, the parameter structure used to configure the filter, and the number of input lines read by the filter (kernel height) in order that the filter may execute once. Note this is not the same as producing a single layer of output.

Filter	Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
DMA	SIPP_DMA_ID	N/A	N/A	DmaParam	1
Mipi Tx	SIPP_MIPI_TX0_ID SIPP_MIPI_TX1_ID	U8, U16, U24, U32, 10P32	N/A	MipiTxParam	N/A
Mipi Rx	SIPP_MIPI_RX0_ID SIPP_MIPI_RX1_ID SIPP_MIPI_RX2_ID SIPP_MIPI_RX3_ID	N/A	U8, U16, U24, U32, 10P32	MipiRxParam	N/A
Sigma Denoise	SIPP_SIGMA_ID	U8, U16	U8, U16	SigmaParam	5
Lens Shading	SIPP_LSC_ID	U8, U16 (Image) U8.8 (Mesh)	U8, U16	LscParam	1
Raw	SIPP_RAW_ID	U8, U16	U8, U16	RawParam	1/3/5
Debayer	SIPP_DBYR_ID	U8, U16	U8, U16	DbyrParam	11
Gen Chroma	SIPP_CGEN_ID	U8, U16	U8	GenChrParam	6
Sharpen	SIPP_SHARPEN_ID	U8F, FP16	U8F, FP16	UsmParam	3/57
DoG/LTM	SIPP_DOG_ID	U8F, FP16	U8F, FP16	DogLtmParam	3/57/9/11/13/15
Luma Denoise	SIPP_LUMA_ID	U8F, FP16	U8F, FP16	YDnsParam	11
Chroma Denoise	SIPP_CHROMA_ID	U8	U8	ChrDnsParam	3
Median	SIPP_MED_ID	U8	U8	MedParam	3
Polyphase FIR	SIPP_UPFIRDN_ID	U8, FP16	U8, FP16	PolyFirParam	7
LUT	SIPP_LUT_ID	U8, U16, FP16	U8, U16, FP16	LutParam	1
Edge operator	SIPP_EDGE_OP_ID	U8	U8, U16	EdgeParam	3
Harris Corners	SIPP_HARRIS_ID	U8	FP16, FP32	HarrisParam	5/7/9
Convolution	SIPP_CONV_ID	U8F, FP16	U8F, FP16	ConvParam	3/5
Color combination	SIPP_CC_ID	FP16,U8 (luma), U8 (chroma)	U8F, FP16 (luma), U8 (chroma)	ColCombParam	1 (Lum), 5(Chr)

Table 9: Summary of MA2150 Hardware Filters

Filter	Filter ID	Buffer	Input buffer ID	Output buffer ID
		Primary buffers		
Sigma denoise	0	in/out	0	0
LSC	1	in/out	1	1
RAW	2	in/out	2	2
Bayer demosaic	3	Bayer in/RGB out	3	3
DoG/LTM	4	in/out	4	4
Luma denoise	5	in/out	5	5
Sharpening	6	in/out	6	6
Chroma generation	7	RGB in/Chroma out	7	7
Median	8	in/out	8	8
Chroma denoise	9	in/out	9	9
Color combination	10	Luma in/RGB out	10	10
Lookup table	11	in/out	11	11
Edge operator	12	in/out	12	12
Convolution kernel	13	in/out	13	13
Harris corners	14	in/out	14	14
Polyphase scaler[0]	15	in/out	15	15
Polyphase scaler[1]	16	in/out	16	16
Polyphase scaler[2]	17	in/out	17	17
MIPI Tx[0]	18	in	18	N.A.
MIPI Tx[1]	19	in	19	N.A.
MIPI Rx[0]	20	out	N.A.	20
MIPI Rx[1]	21	out	N.A.	21
MIPI Rx[2]	22	out	N.A.	22
MIPI Rx[3]	23	out	N.A.	23
		Secondary buffers		
LSC	1	Gain mesh buffer	20	N.A.
Median	8	Luma in	21	N.A.
Color combination	10	Chroma in	22	N.A.
Lookup table	11	LUT buffer	23	N.A.
Luma denoise	5	Cosine 4th law LUT buffer	24	N.A.
Color combination	10	3D LUT buffer	25	N.A.
RAW	2	Defect list	26	N.A.
Bayer demosaic	3	Luma out	N.A.	18
RAW	2	AE statistics	N.A.	19
RAW	2	AF statistics		24
RAW	2	Luma histogram		25
RAW	2	RGB histogram		26

Table 10: Filter IDs and input/output buffer IDs

8.1 DMA

The DMA filter can be used to transfer image data from DDR to CMX, and vice versa. An instance of the DMA filter must be either a source filter or a sink filter It either sources data from DDR as an input to the pipeline, or writes processed data out to DDR as an output from the pipeline. The transfer of data is either from DDR to the DMA filter's output buffer (DMA filter is a source) or from the DMA filter's parent's output buffer to DDR (DMA filter is a sink).

DMA transfers have a source and a destination. The hardware has completely independent state machines for managing the fetching of source data vs. managing the storing of destination data. As such, the configuration of source and destination are completely independent.

Both the source and the destination need to be configured according to the layout of the corresponding image in memory. Images may contain padding at the right hand side of the image. A programmable stride allows the padded width of the image to be greater than the actual width. Padding bytes will not be transferred. Additionally, the DMA controller supports the concept of "chunking". Chunking supports transfers to/from Output Buffers where the scanlines are split across multiple slices.

A total of 4 parameters are used to describe an image layout in memory (all units are in bytes):

- Image line width.
- Chunk width.
- Chunk stride.
- Plane stride.

Each of these four parameters are independently programmable for both the source and destination images.

8.1.1 Automatic calculation of image layout parameters

An automatic mode is supported, whereby the framework calculates the image layout parameters internally. This mode should be used when the image is a SIPP-managed buffer (i.e. an Output Buffer located in CMX). For a source filter, the destination image should use automatic mode, and for a sink filter, the source image should use automatic mode. To specify the use of automatic mode, the "Chunk width" parameter should be set to 0.

8.1.2 Images in DDR

For images in DDR, the filter needs to be configured with the image stride (“Line Stride”). If the image has more than one plane, a plane stride must also be specified. The Chunk Width should be set to the number of bytes of data to be transferred per line. Since data in DDR is normally never split into slices, the Chunk Stride should be identical to the Chunk Width.

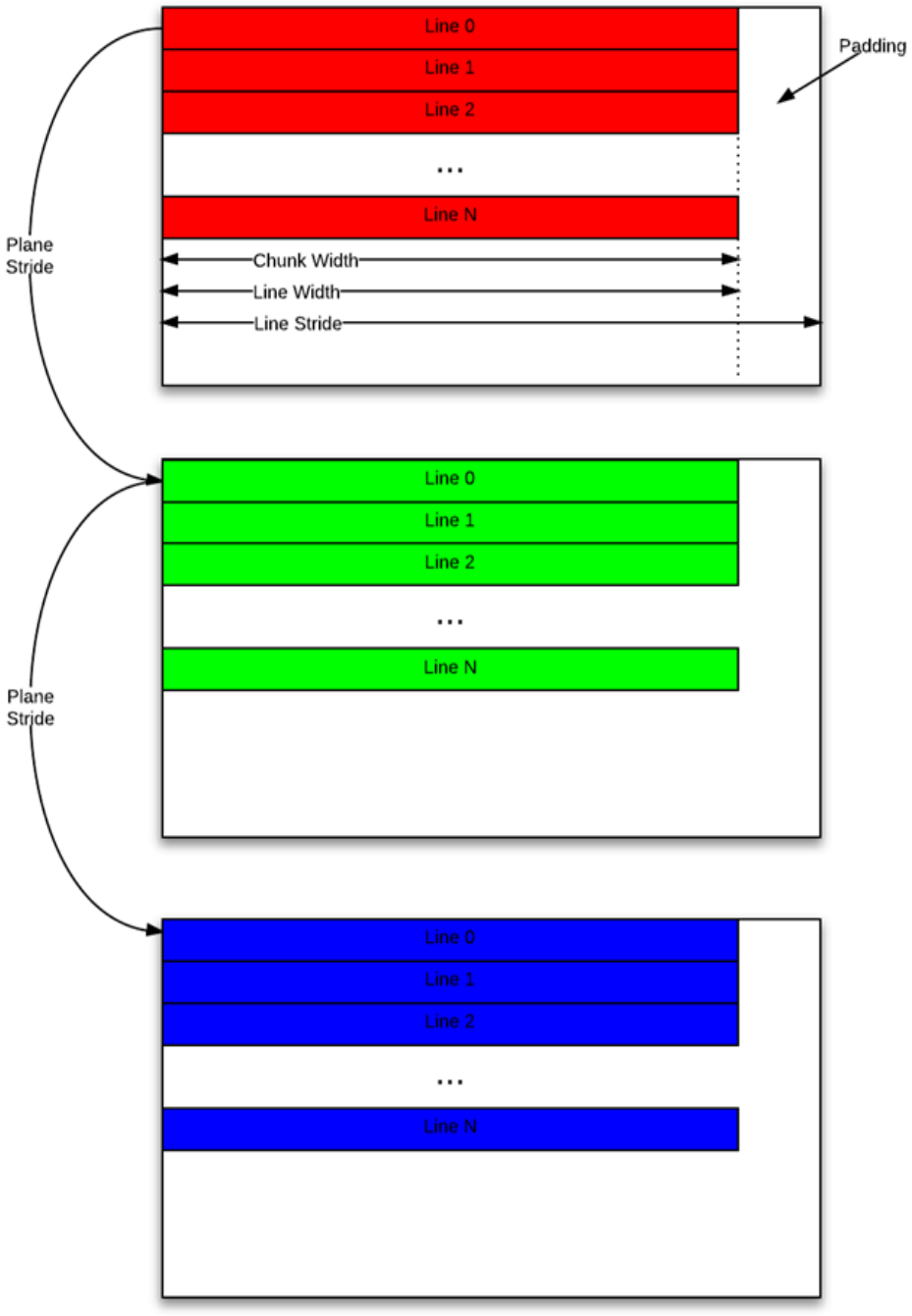


Figure 15: DMA configuration targeting image in DDR

8.1.3 Images in CMX

For Images in CMX, Automatic Mode should be used. Set the Chunk Width parameter to 0 to enable automatic mode. The configuration parameters (shown below) will be calculated automatically.

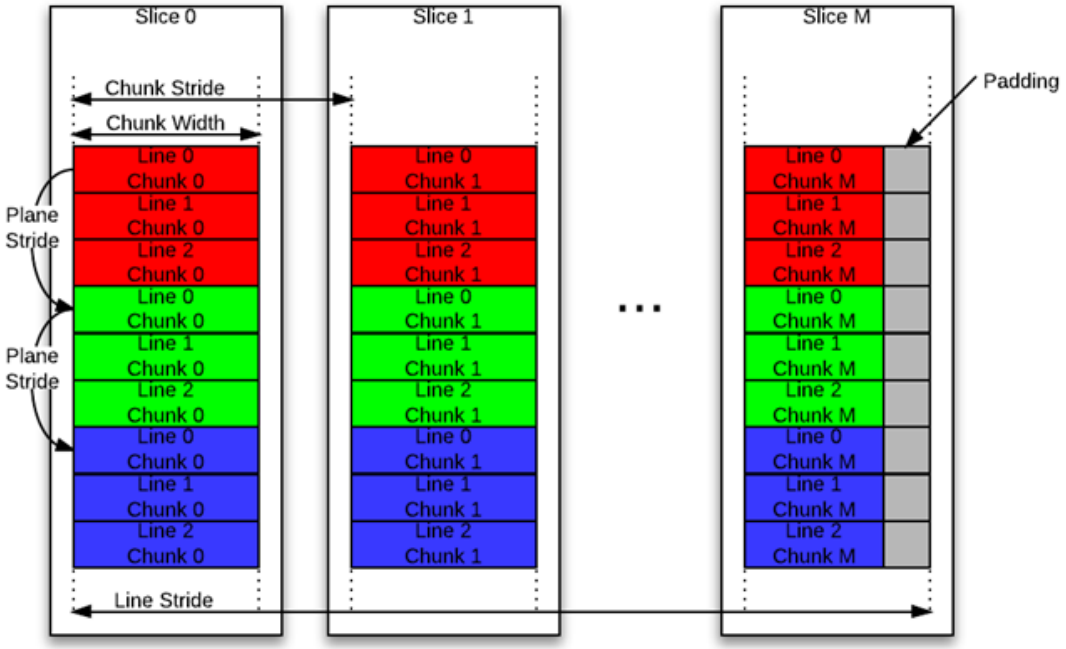


Figure 16: DMA configuration targeting sliced image in CMX memory

8.1.4 DmaParam Configuration

This filter is configured via the ***DmaParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
ddrAddr	31:0	DDR memory address of the image in DDR. If the transfer is from DDR to the DMA filter's output buffer, this is the source image address. If the transfer is from the parent filter's output buffer to DDR, this is the destination image address.
dstChkW	31:0	Chunk Width of the destination image
srcChkW	31:0	Chunk Width of the source image
dstChkS	31:0	Chunk Stride of the destination image
srcChkS	31:0	Chunk Stride of the source image
dstPlS	31:0	Plane Stride of the destination image
srcPlS	31:0	Plane Stride of the source image
dstLnS	31:0	Line Stride of the destination Image
srcLnS	31:0	Line Stride of the source Image

8.2 MIPI Rx

Input	Formatted MIPI CSI-2/DSI data via MIPI controller *hsync/vsync/valid/data* parallel interface
Operation	Flexible stream processing of input directly from MIPI RX including active image region windowing, sub-sampling, data-selection, array sensor plane extraction, black level subtraction (for RAW input), and data format conversion.
Input buffer	None – input streams from MIPI RX controller parallel interface.
Output	Up to 16 planes of - U8/U16/U32/10O32 - Packed RGB888 (3 bytes) - Packed YUV888 (3 bytes) RAW/YCbCr/RGB in up to 4 planes
Instances	4

Table 11: MIPI Rx filter overview

The MIPI Rx filters connect to MIPI Rx channels via a parallel interface and adapt the *hsync*, *vsync*, *valid* and *data* signals (shown in Table 12) from the MIPI controller to drive their internal data path.

Signal	Bits	Direction	Description
VSYNC	1	Input	Vertical synchronization signal
HSYNC	1	Input	Horizontal synchronization signal
DATA	32	Input	Input data interface
VALID	1	Input	Input data valid

Table 12: MIPI controller Rx interface (signal directions relative to SIPP

Data is clocked into the MIPI Rx filters using the media clock. The media clock is synchronous to the SIPP system clock but may run either at full speed, half speed or quarter speed. The MIPI Rx data may therefore be transferred to the SIPP system clock domain without any special synchronization.

Figure 17 shows a block diagram of the MIPI Rx filter. The filter architecture defines a simple yet flexible approach to the formatting and storage of incoming image data. The filter is primarily intended to handle RAW data or single channels/components of YUV/YCbCr/RGB. There is no support for scattering of separate components/channels of YUV/YCbCr/RGB to multiple buffers or planes.

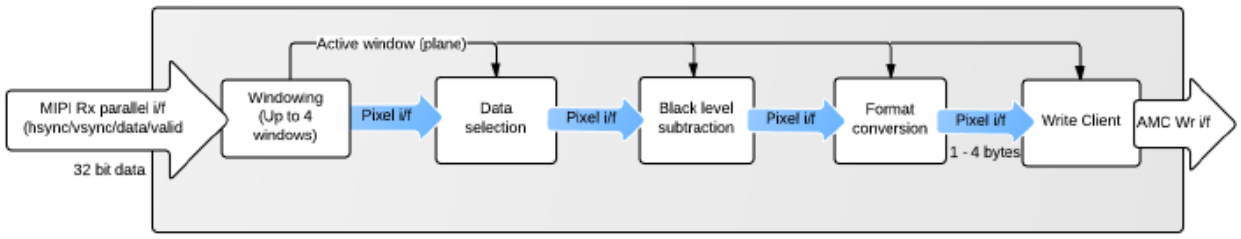


Figure 17: MIPI Rx filter block diagram

8.2.1 MIPI Rx Features

8.2.1.1 Windowing

The incoming data stream from the MIPI controller may be windowed, meaning that only pixels falling within a defined window (or windows) is forwarded for formatting/storage (i.e. an image crop may be affected).

A maximum of 4 orthogonal (non-overlapping) windows may be used allowing, for example, RGBW array sensor data or mixed data-type frames to be output to separate planes of a planar buffer.

The window grid is defined in terms of a set of (x, y) co-ordinates. There are 4 x co-ordinates (x_0, x_1, x_2 and x_3) and 4 y co-ordinates (y_0, y_1, y_2 and y_3) which define the top-left corners of up to 4 windows in any of the following shapes (number horizontally x number vertically):

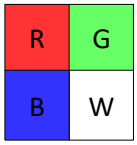
- Single window: 1x1
- Single row of windows: e.g. 2x1, 3x1 or 4x1
- Single column of windows: e.g. 1x2, 1x3 or 1x4
- 2x2 array of windows

Each of the windows in a single column must have the same width but windows in a single row may have different widths. There 4 programmable window widths giving the widths of the windows starting at x_0, x_1, x_2 and x_3, respectively. For windowed output to a single plane the total of the widths for the all windows in a row must be equal to the width of the output frame. However, for output to multiple planes of a planar buffer the width of each window must be the same and must be equal to the width of the output frame.

Each of the windows in a single row must have the same height but windows on different rows may have different heights; there are 4 programmable window heights giving the heights of all windows starting at y_0, y_1, y_2 and y_3, respectively. For windowed output to a single frame the total heights for all the windows in a column must be equal to the height of the output frame.

The filter tracks the input x, y co-ordinate by incrementing two counts as data are received on the parallel interface. The counts are compared to the programmed windows to determine which window is active.

Windows may also be interleaved. For example, data from an array sensor may be scanned-in in *interleaved raster order*. That is, where the sensor is organized as:



Then the first line received will span the R/G (at the top of the sensor) and the second line received will span the B/W (at the bottom of the sensor); thus lines from R/G and B/W are interleaved. In fact, multiple lines from the top may be followed by multiple lines from the bottom. The bit of the filter's line count (y co-ordinate) which is used to determine whether the filter is receiving for the top or bottom set of windows is therefore programmable.

Figure 18 shows an example 2x2 window grid. The windows are defined as follows:

- Windows (0,0) and (1,0) share the same start x co-ordinates (x_0 and x_1)
- Windows (0,1) and (1,1) share the same start x co-ordinates (x_2 and x_3)
- Windows (0,0) and (0,1) share the same start y co-ordinates (y_0 and y_1)
- Windows (1,0) and (1,1) share the same start y co-ordinates (y_2 and y_3)

If interleaved mode is configured then Windows (0,0) and (0,1) (or (0,1) and (1,1)) can use exactly the same co-ordinates since the co-ordinates are relative to the top/bottom halves (or fields) of the frame.

The following general restrictions apply to the specification of the window grid (unless in interleaved mode) to ensure that no windows overlap:

$x_i < x_{i+1}$

$width_i <= (x_{i+1} - x_i)$

$y_i < y_{i+1}$

$height_i <= (y_{i+1} - y_i)$

All windows in a column *must* have the same width and all windows in a row must have the same height.

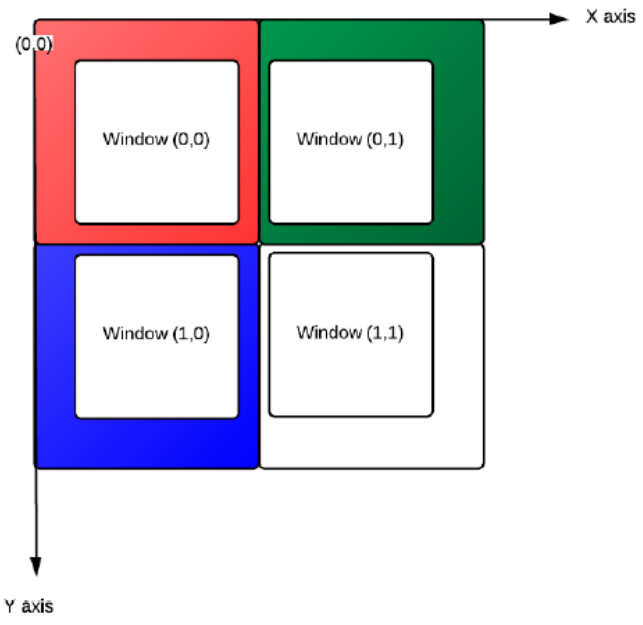


Figure 18: Example MIPI Rx window grid for RAW data from RGBW array sensor

Data falling in different windows may (optionally) be stored in separate planes of a planar buffer. The number of planes of output corresponds to the number of windows in use – a maximum of 4 are supported. The plane stride **ps** and number of planes **np** for the output buffer should be programmed appropriately to

match the size and number of windows. For planar output the plane indices correspond to the windows as they occur in raster (or interleaved raster) order from left to right and top to bottom in the incoming frame.

8.2.1.2 Data selection and mask

The input parallel interface has a 32 bit data bus. The least significant bit at which the data selection starts is programmable. A programmable 32 bit mask is provided which is ANDed with the right-shifted selection allowing, for example, only the luma component to be picked off from the bus. For example if the incoming data is *d*, the selection bit is *b* and the mask is *m* then the selection *s* is given by:

```
s = (d >> b) & m;
```

By setting the selection bit to zero and the mask to 0xffffffff the full data bus may be selected. If black level selection and format conversion are disabled for the window then the full data bus may be written to memory unmodified.

For further flexibility selection may enabled/disabled for even/odd pixels and even/odd lines. If selection is disabled it means that that pixel (or line) is not output but is skipped over; care must be taken to adjust the width/height of the output to match appropriately.

8.2.1.3 Black level subtraction

For RAW input black level subtraction may be performed. Four programmable 16 bit black levels are available: $black_0$, $black_1$, $black_2$ and $black_3$. For Bayer input $black_0$/$black_1$ are used on even/odd pixels on even lines and $black_2$/$black_3$ are used on even/odd pixels on odd lines. For planar (array sensor) input $black_0$, $black_1$, $black_2$ and $black_3$ are used for planes 0, 1, 2 and 3 respectively. Black level subtraction may be enabled individually for each window. The result of the subtraction is checked for underflow; any result less than zero is clamped to zero.

8.2.1.4 Format conversion

n the final stage input data with a precision higher than 8 bits may be down converted to 8 bits. The down conversion is performed by right-shifting the input data by a programmable number of bits then rounding by adding the bit value in the position below bit 0 (after right-shift). The result is saturated to 8 bits. A single right-shift (number of bits) for format conversion is programmable for all windows, but format conversion may be enabled individually for each window.

8.2.1.5 Buffer

As there no back pressure can be applied back to the MIPI if the write client become full. A 256 entry fifo is placed between the Format conversion block and the write client. This will help protect against momentary stalls on the write client interface. Additionally if the output data after format conversion is 16 bits or less this fifo will have the data packed such that there will be 512 entries.

8.2.1.6 Output

The final output data may be in 1-4 bytes. As such the format bit field of the filter's SIPPOBuf[N]Cfg register, which indicates the size of the data in bytes, should be set appropriately. Note that the same format applies to all planes of a planar buffer so if a mixed data-type frame is being windowed and output to separate planes then the same number of bytes per pixel and pixels per line must be output for each plane.

8.2.2 MIPI Rx Configuration

Name	Bits	Description
frmDim		PRIVATE – frame dimensions in pixels
	15:0	Frame width
	31:16	Frame height
cfg	1:0	Clock Speed 00 – Clk MIPI = Clk MIPIRX 01 – Clk MIPI = (Clk MIPIRX)/2 10 – Clk MIPI = (Clk MIPIRX)/4
	2	Reserved
	7:4	Reserved
	11:8	Format conversion enables (per window)
	16:12	Format conversion right-shift (number of bits)
	17	Bayer/planar configuration (for black level subtraction) 0 – Planar (array sensor) input 1 – Bayer input
	18	Output configuration 0 – Data from each window written to separate plane 1 – Data from all windows written to single plane (frame)
	19	Pack Buffer If formatted data is 16 bits or two pixels may be packed into each entry of the buffer FIFO.
	23:20	Black level subtraction enable (per window)
	24	Use packed windows
	25	Use private chunk stride
	26	Promote data from input bit depth to 16 bit
	31:28	Input bit depth; Value programmed should be bit depth -1
winX[4]	15:0	x_start – x co-ordinate at which window starts
	31:16	x_width – width N^{th} window (N=0..3) or if packed windows are enabled specifies the width of each window
winY[4]	15:0	y_start – y co-ordinate at which window starts
	31:16	y_height – h_0, height of N^{th} window (N=0..3)
sel01	4:0	Least significant bit of Window 0 selection
	11:8	Selection enable (set to 1 to enable, if not enabled pixel is skipped over) Bit 0 – even pixels on even lines Bit 1 – odd pixels on even lines Bit 2 – even pixels on odd lines Bit 3 – odd pixels on odd lines
	19:15	Least significant bit of Window 1 selection
	27:24	Selection enable (set to 1 to enable, if not enabled pixel is skipped over)

Name	Bits	Description
		Bit 0 – even pixels on even lines Bit 1 – odd pixels on even lines Bit 2 – even pixels on odd lines Bit 3 – odd pixels on odd lines
sel23	4:0 11:8 19:15 27:24	Least significant bit of Window 2 selection Selection enable (set to 1 to enable, if not enabled pixel is skipped over) Bit 0 – even pixels on even lines Bit 1 – odd pixels on even lines Bit 2 – even pixels on odd lines Bit 3 – odd pixels on odd lines Least significant bit of Window 3 selection Selection enable (set to 1 to enable, if not enabled pixel is skipped over) Bit 0 – even pixels on even lines Bit 1 – odd pixels on even lines Bit 2 – even pixels on odd lines Bit 3 – odd pixels on odd lines
selMask[4]	0:31	Selection mask for window N (N=0..3)
black01	 15:0 31:16	Black levels 0 and 1 $black_0$ – for Window 0 or Even pixel on even line for Bayer data $black_1$ – for Window 1 or Odd pixel on even line for Bayer data
black23	 15:0 31:16	Black levels 2 and 3 $black_2$ – for Window 2 or Even pixel on even line for Bayer data $black_3$ – for Window 3 or Odd pixel on even line for Bayer data
vbp	15:0 31:16	Vertical black porch. Specifies vertical back porch in lines (not normally required) Private Chunk Stride. Must be a multiple of 8 bytes.

8.3 MIPI Tx

Input	Up to 16 planes (sequentially) of - U8/U16/U32 - Packed RGB888 (3 bytes) - Packed YUV888 (3 bytes)
Operation	Timing generation for MIPI *Tx* controller parallel interface for CSI-2/DSI output
Input buffer	Minimum of 1 line
Output	MIPI CSI-2/DSI output via MIPI *Tx* controller parallel interface
Instances	2

Table 13: MIPI *Tx* filter overview

8.3.1 MIPI Tx Configuration

Name	Bits	Description
frmDim		PRIVATE - frame dimensions in pixels
	15:0	Frame width
	31:16	Frame height
cfg	0	Scan Mode 0 – Progressive scan mode 1 – Interlace scan mode
	1	First Field 0 – Use standard timing configuration settings for first field 1 – Use even timing configuration settings for first field
	2	Display Mode 0 – continuous 1 – one shot mode
	3	Level of HSYNC/VSYNC when timing FSMs are in IDLE state
	5:4	Media Clock Speed 00 – MIPI pixel clock = SIPP clock (this configuration is illegal) 01 – MIPI pixel clock = SIPP clock/2 10 – MIPI pixel clock = SIPP clock/4
		Vertical interval in which to generate vertical interval

Name	Bits	Description
	7:6 8 9	interrupt 00 – VSYNC 01 – Back Porch 10 – Active 11 – Front Porch Level of VSYNC when timing FSM is in BACKPORCH state Level of VSYNC when timing FSM is in FRONTPORCH state
lineCompare	31:0	Line count at which to generate line compare interrupt
vCompare	31:0	Vertical interval in which to generate vertical interval interrupt 00 – VSYNC 01 – Back Porch 10 – Active 11 – Front Porch
hSyncWidth	31:0	Specifies the width, in PCLK clock periods, of the horizontal sync pulse (value programmed is HSW-1)
hBackPorch	31:0	Specifies the number of PCLK clocks from the end of the horizontal sync pulse to the start of horizontal active (value programmed is HBP so a back porch of 0 cycles can be set)
hActiveWidth	31:0	Specifies the number of PCLK clocks in the horizontal active section (value programmed is AVW-1)
hFrontPorch	31:0	Specifies the number of PCLK clocks from end of active video to the start of horizontal sync (value programmed is HFP)
vSyncWidth	31:0	Specifies the width in lines of the vertical sync pulse (value programmed is VSW-1) This value is used for the odd field when in interlace mode
vBackPorch	31:0	Specifies the number of lines from the end of the vertical sync pulse to the start of vertical active (value programmed is VBP) This value is used for the odd field when in interlace mode

Name	Bits	Description
vActiveHeight	31:0	Specifies the number of lines in the vertical active section (value programmed is AVH-1) This value is used for the odd field when in interlace mode
vFrontPorch	31:0	Specifies the number of lines from the end of active data to the start of vertical sync pulse (value programmed is VFP) This value is used for the odd field when in interlace mode
vSyncStartOff	31:0	Number of PCLKs from the start of the last horizontal sync pulse in the Vertical Front Porch to the start of the vertical sync pulse. This value is used for the odd field when in interlace mode
vSyncEndOff	31:0	Number of PCLKs from the end of the last horizontal sync pulse in the Vertical Sync Active to the end of the vertical sync pulse. This value is used for the odd field when in interlace scan mode

8.4 Sigma Denoise

Input	Up to 16 bit RAW (Bayer pattern or non Bayer data)
Operation	Weighted averaging filter primarily intended to operate in the Bayer domain
Filter kernel	5x5
Local line buffer	Yes, maximum supported image width is 4624
Output	Up to 16 bit RAW (Bayer pattern or non Bayer data)
Instances	1

The Sigma denoise filter is a weighted averaging filter. It is primarily intended to operate in the Bayer domain, but can also operate on non-Bayer data. There are 2 programmable threshold multipliers, called T_1 and T_2, which can be adapted according to the SNR ratio of the image. A separate pair of threshold multipliers may be specified for each of the 4 Bayer channels. There is also a programmable noise floor, which can be adapted according to the noise characteristics of the camera sensor.

This filter replaces each pixel with a weighted average of 9 pixels in the neighborhood. In Bayer mode, the filter works over a 5x5 neighborhood, but only averages pixels in the same Bayer channel. The Gr and Gb channels are processed independently. In non-bayer mode, the filter works over a 3x3 neighborhood. So in all cases, a total of 9 pixels are averaged. Each pixel to be averaged is assigned a weight of either 0, 1 or 2. Pixels assigned a weight of 0 have no impact on the result. The weighted average, V', is calculated as follows:

$$V' = \frac{\sum_{i=1}^{9} W_i \cdot V_i}{\sum_{i=1}^{9} W_i}$$

where V_i are the pixel values in the neighborhood to be averaged (including the center pixel), and W_i are the weights assigned to the corresponding pixels. The weights are calculated by computing the absolute differences between the center pixel and the neighborhood pixels, and comparing the result against thresholds. The thresholds are calculated internally based on a noise model which estimates the variance of the noise component of the signal, and are then multiplied by the user-programmable threshold multipliers T_1 and T_2. If the absolute difference is less than or equal to both thresholds, the pixel is assigned a weight of 2. If the absolute difference is less than or equal to only one of the thresholds, the pixel is assigned a weight of 1. If the absolute difference is greater than both thresholds, the pixel is assigned a weight of zero. In this way, the filter will not mix dissimilar pixels, thus avoiding blurring of edges and suppressing of detail.

The two thresholds to compare against, TH_1 and TH_2, are computed as follows:

$$N = max(NF, \sqrt{(L)})$$
$$TH_1 = N \cdot T_1$$
$$TH_2 = N \cdot T_2$$

where NF is a user-programmable Noise Floor, L is the pixel intensity level in the neighborhood, T1 and T2 are the user-programmable thresholds, and N is an estimate of the noise signal variance as a function of the neighborhood pixel intensity.

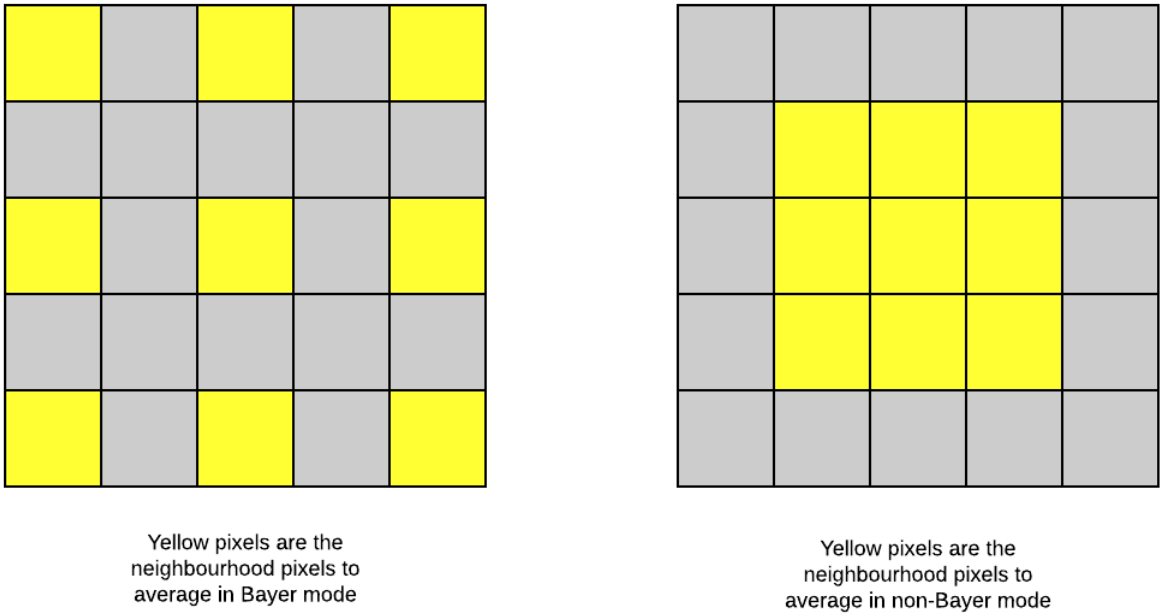
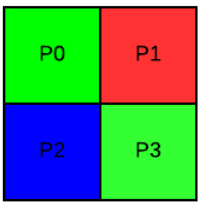


Figure 19: Bayer versus non-Bayer mode of operation

From the four programmed sets of threshold multipliers, one set (T_1 and T_2) is selected based on the Bayer Position of the pixel. If the image were divided into 2x2 blocks, a pixel could be in one of 4 positions within a block. How the Bayer position maps to a given color channel (Gr, R, B or Gb) depends on the Bayer order of the image. When programming the threshold registers, this must be taken into account. The illustration below shows how pixel positions map to Bayer color channels, in the case where the Bayer Order is GRBG. If the Bayer Order is different, then the mapping of pixel positions to color channels will also be different.



8.4.1 Additional features

8.4.1.1 Black level Subtraction

Black level subtraction may be performed on the filtered pixels by programming `SigmaParam::blcGR`, `SigmaParam::blcR`, `SigmaParam::blcB` & `SigmaParam::blcGB` variables appropriately. (Setting black levels to 0 disables black level subtraction.)

8.4.2 Configuration

The Sigma Denoise filter is configured via the ***SigmaParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
FrmDim	31:16	Frame height
	15:0	Frame width
thresh[0]	31:24	Bayer position 1 – threshold multiplier 1 U(0,8)
	23:16	Bayer position 1 – threshold multiplier 0 U(0,8)
	15:8	Bayer position 0 – threshold multiplier 1 U(0,8)
	7:0	Bayer position 0 – threshold multiplier 0 U(0,8)
thresh[0]	31:24	Bayer position 3 – threshold multiplier 1 U(0,8)
	23:16	Bayer position 3 – threshold multiplier 0 U(0,8)
	15:8	Bayer position 2 – threshold multiplier 1 U(0,8)
	7:0	Bayer position 2 – threshold multiplier 0 U(0,8)
cfg	23:8	Noise floor – minimum noise variance (noise variance when there is no incident light on the sensor). The range is the same as the range of the incoming pixel range, as given by the Data Width in Bits field.
	7:4	Data pixel width – 1
	1	Enable pass-through mode
	0	RAW data format: 0 – Planar data 1 – Bayer data
bayerPattern	1:0	Bayer pattern (00:GRBG, 01:RGGB, 02:GBRG, 03:BGGR
blcGR	15:0	Bayer mode: black level for GR Pix Planar mode: black level for plane 0, plane 4, plane 8, plane 12 etc
blcR	15:0	Bayer mode: black level for R Pix Planar mode: black level for plane 1, plane 5, plane 9, plane 13 etc
blcB	15:0	Bayer mode: black level for B Pix Planar mode: black level for plane 2, plane 6, plane 10, plane 14 etc
blcGB	15:0	Bayer mode: black level for GB Pix Planar mode: black level for plane 3, plane 7, plane 11, plane 15 etc

8.5 Raw Filter

Input	Up to 14 bit RAW (Bayer pattern or non Bayer data)
Operation	Gr/Gb imbalance correction Hot pixel suppression Digital gain and saturation Patch-based colour channel/plane accumulation statistics (AE/AWB) Patch-based AF stats Histograms
Filter kernel	5x5
Local line buffer	Yes, maximum supported image width is 4624
Output	Up to 14 bit RAW (Bayer pattern or non Bayer data) + statistics and histograms
Instances	1

The RAW filter can handle either Bayer pattern or non-Bayer data where each color channel is stored in a different plane of a planar buffer.

For Bayer data the first stage of processing is Gr/Gb imbalance and, in parallel, bad kernel detection/defect pixel correction. These processing stages use a 5x5 pixel kernel. Generally the output of Gr/Gb imbalance is forwarded to the next stage: digital gain and saturation. However, if a defect pixel is detected, it is corrected and forwarded instead. Defect pixel correction may be configured to touch only green (Gr and Gb) pixels, leaving R and B pixels unmodified.

For non Bayer data Gr/Gb imbalance (including bad kernel detection) should not be enabled. If a defect pixel is detected then it is corrected and forwarded to digital gain and saturation.

The bit depth of the input RAW may be between 6 and 16 bits. The input data width (in bytes) is programmable for the RAW filter. If the bit depth is 8 bits or fewer the input buffer must be organized with the values of each color channel packed into the LS bits of each byte. If the bit depth is greater than 8 bits the input buffer must be organized with the values of each color channel packed into the LS bits of each pair of bytes.

8.5.1 Features

8.5.1.1 Statistics

There are four categories of statistics which may be independently enabled:

- Statistics suitable for AE/AWB (Auto Exposure / Auto-White-Balance) algorithms.
- Statistics suitable for AF (Auto Focus) algorithms.
- 256-bin Luma Histogram.
- 128-bin RGB histogram.

For each statistics type, there is an associated output buffer. Each output buffer has a corresponding base address register (see Table 10). There is no programmable stride, format etc. is fixed, and the usual corresponding configuration registers are not implemented. As usual the base addresses of the buffers should be aligned on 64 bit boundaries.

For Bayer data the AE/AWB statistics are gathered on a single designated plane (if processing multiple planes) or on four designated planes for non-Bayer data. The AF statistics are gathered on a single designated plane only, irrespective of whether the data is Bayer or non-Bayer.

Luma Histogram statistics are gathered on a Luma channel derived from input Bayer data or on the designated white or clear channel (plane) for non-Bayer. RGB histogram statistics are gathered on a single designated plane for Bayer data, or on three designated (Red, Green and Blue) planes for non-Bayer data.

Note that statistics are gathered in parallel with defect correction and are based on the uncorrected input data.

8.5.1.2 Static defect pixel correction

Static defect pixel correction works by reading a defect list from CMX, via the RAW filter's defect input buffer. This input buffer's base address register should be set to point to the defect list (see Table 10). The defect list is a one dimensional array of 32-bit entries, one entry per defect (though whole defective rows may be signaled with a single entry using a special marker, described below). There is no programmable stride, format is fixed, and the usual corresponding configuration registers are not implemented. As usual the BASE address of the buffer should be aligned on a 64 bit boundary. Each entry in the defect list contains the plane and (X, Y) location in the image of where the defective pixel is located, as shown in Table 14. The list must be sorted to match the order in which the defective pixels will be encountered when processing the image (raster order). Note that a maximum of 8 defective pixels may be corrected in a given scanline.

Bit field	Definition
31	Red/Blue correction direction bit 0 – Horizontal correction 1 – Vertical correction
30:27	Plane number
26:14	Y co-ordinate of defective pixel
13:0	X co-ordinate of defective pixel

Table 14: Defect list entry bit fields

Defective pixels of the Green channel of a Bayer image will be replaced by simple averaging of the 4 nearest green pixels (the four diagonal neighbors). In the case of the Red or Blue channels, the defective pixel is replaced by the average of two of the four nearest pixels in the same color channel. The correction direction bit in the defect list entry controls whether Red and Blue pixels are to be corrected by averaging the two horizontal neighbors in the same color channel, or the two vertical neighbors in the same color channel.

Note that 14 bits are provided to specify the X co-ordinate but only 13 bits are provided to specify the Y co-ordinate. The maximum values for X and Y are used to signal extra information as follows:

- If (X, Y) is equal to (0x3FFF, 0x1FFF) this marks the end of the list, and no further entries will be read.
- If the X co-ordinate is 0x3FFF but the Y co-ordinate is not 0x1FFF, then this signals a bad row of pixels; all pixels on row Y will be corrected.

To correct an entire row, the correction direction bit should be set to 1 (vertical correction) in the list entry signaling the bad row. To correct an entire column of pixels, an explicit entry in the defect list must be present for every pixel in the column; if the image height is 1080 then 1080 list entries are required to perform correction of that column. The correction direction bit should be set to 0 (horizontal correction) in each list entry pertaining to the bad column.

The defect list is read from the defect input buffer as required by the filter. The buffer is accessed via the filter's read client interface, the same read client interface as used to fill the filter's local line buffer. The first few entries of the defect list are pre-loaded into a FIFO within the filter before each frame starts, the rest of the list is read in as defects are processed and space becomes available in the FIFO. These accesses are only performed when the read client interface is otherwise idle, typically for a few cycles between each line. If the filter is used within the oPipe with data streaming directly into it then the read client interface always available exclusively for defect list read access. Note that the first entries of the defect list will be read as soon as the filter is enabled and static defect correction is enabled, the defect list buffer base address should be programmed correctly before the corresponding enable bits are set.

8.5.1.3 Dynamic defect pixel correction

Dynamic pixel correction works by analyzing local gradients, and determining if the pixel is an outlier with respect to the pixels in the 5x5 neighborhood. Both hot (bright) and cold (dark) pixels can be corrected. If the pixel is determined to be an outlier, it is reduced in value (in the case of hot pixels) or increased in value (in the case of cold pixels) in proportion to the local gradient magnitudes. The aggressiveness of the filter can be independently controlled for hot a cold pixels, and for Green pixels vs. Red/Blue pixels. Additionally, there is a noise threshold which may be increased to increase the overall aggressiveness of the filter. This works by subtracting a noise threshold from the pixel values, before the gradients are computed.

8.5.1.4 Gr/Gb imbalance correction, bad kernel detection and defect pixel correction

The processing in the RAW filter is performed using optimized fixed-point arithmetic. The format of the input data is U(W, 0) where W is specified via the *RawParam::cfg* variable.

Gr/Gb imbalance correction is used to correct the imbalance between Gr (green pixel horizontally adjacent to red) and Gb (green pixel horizontally adjacent to blue) in a Bayer image. This imbalance correction can be distorted by hot or cold pixels (bad pixels) in the locality (i.e. a bad kernel) and will effectively leave a “scar” around the site where a bad pixel resides. In order to allow the Gr/Gb imbalance correction and defect pixel correction filtering to operate in parallel without the Gr/Gb imbalance filter introducing these “scar” artifacts a bad kernel detection filter is defined. This filter detects when a bad green pixel is in the 5x5 locality and effectively disables Gr/Gb imbalance correction for the current active green pixel.

All three of these filters use the same 5x5 kernel and run in parallel. Gr/Gb imbalance, bad kernel detection and bad pixel suppression are then combined as follows: if a bad kernel is detected then the Gr/Gb imbalance filter will perform no adjustment to the data and will output the original data instead; however, if the current pixel (center pixel of the kernel) is detected as a defect pixel then it is corrected, based on the original input data, and the corrected pixel is output instead of the Gr/Gb imbalance corrected pixel.

The filter does not process the first and last two rows and columns of the input image; these are copied unmodified to the output.

8.5.1.5 Digital gain and saturation

The final RAW processing step is the application of a digital gain. Four 16 bit U(8,8) gain values and four U16 saturation values are programmable via *Rawparam::gainSat[4].*

There are two selectable modes of operation: 2x2 mode and 4x4 mode.

- 2x2 mode: this mode is for use with a standard 2x2 Bayer CFA. The four registers are mapped to the four positions in a 2x2 Bayer block. These values must be programmed taking the Bayer Order of the data into account as described in the gainSat[4] entry in Table 15.
- 4x4 mode: The four registers are used in sequence, i.e. the programmed values in *Rawparam::gainSat[0]* are applied to pixel 0, the values in *Rawparam::gainSat[1]* are applied to pixel 1, the values in *Rawparam::gainSat[2]* are applied to pixel 2, the values in *Rawparam::gainSat[3]*are applied to pixel 3, the values in *Rawparam::gainSat[0]* are applied to pixel 4 etc.

8.5.1.6 AE/AWB Patch accumulation statistics

If enabled the RAW filter will dump a number of accumulations gathered over a configurable array of patches overlaid on the image and a luma histogram to its statistics output buffer. (Note that like all other buffers this buffer must be 64 bit aligned.)

A 2D array of patches overlaid on the planes of input image may be specified with the RawParam::statsPatchCfg, RawParam::statsPatchStart, RawParam::statsPatchSkip and RawParam::statsPlanes variables. The values of all pixels (of the same color channel for Bayer data) within a patch are accumulated.

The number of patches horizontally and vertically and patch width and height are separately configurable. The maximum number of patches is 64x64. The maximum patch size 256x256. Only one plane of Bayer data can be monitored – the designated plane is programmed via the RawParam::statsPlanes variable. For non Bayer data up to four planes may be monitored, also programmed separately via the RawParam::statsPlanes variable. The patches are specified by their size, number, start location – the (x, y) co-ordinate of the top-left patch in the image – and the distance horizontally/vertically to the next patch in the row/column. (Value programmed for all sizes is size is minus one).

There are two programmable thresholds: a dark threshold and a bright threshold. Pixels that are less than the dark threshold or greater than the bright threshold are accumulated by an alternate set of accumulators. Otherwise, they are accumulated by the primary set of accumulators. Furthermore, when operating in the Bayer domain, if any pixel in a 2x2 block is directed to the alternate set of accumulators, then all pixels in that 2x2 block shall be directed to the alternate set of accumulators. A count of the number of pixels (or 2x2 blocks in the case of Bayer data) which were directed to the alternate set of accumulators is maintained and output per-patch.

The accumulation is performed on only the 8 most significant bits of the input data. The size of the accumulations internally is 24 bits, but they are written out to the statistics output buffer as U32 values.

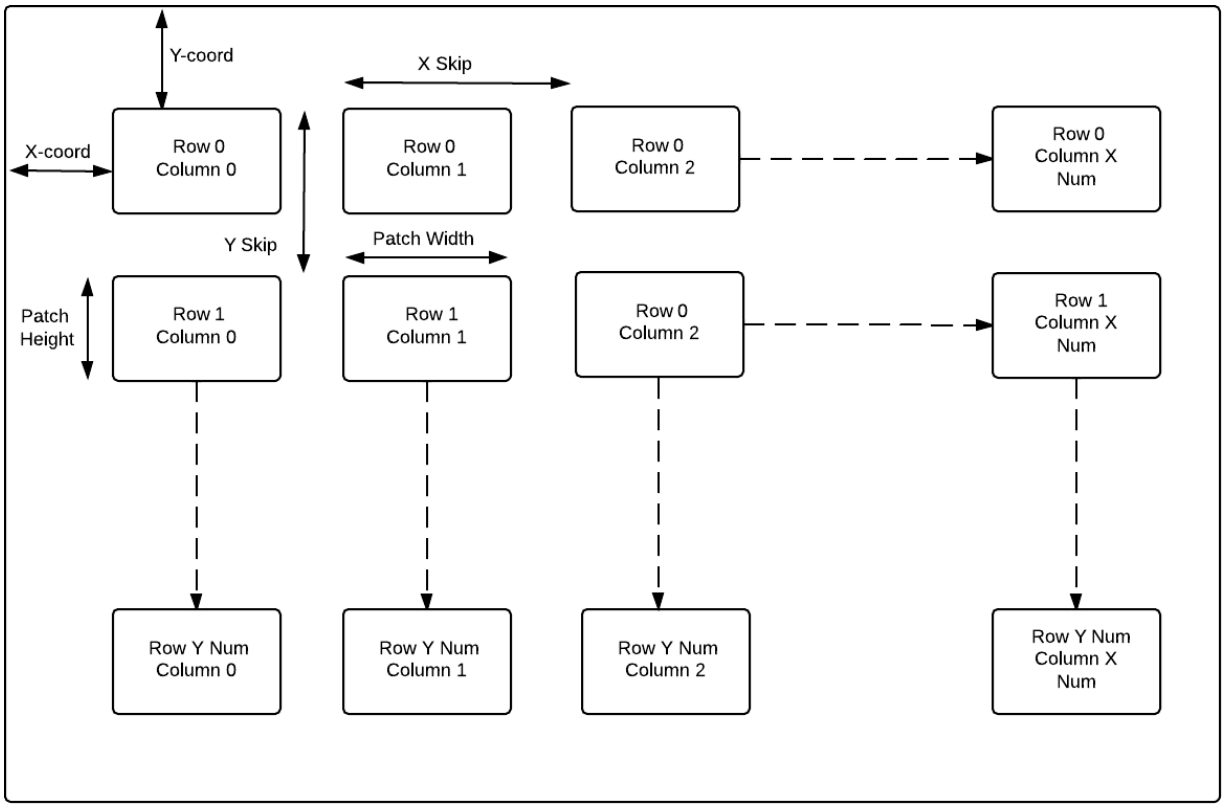


Figure 20: Patch statistics configuration

A local RAM is used to store the partial accumulations as lines scan through the filter. On the last line of a patch and at the end of the patch window in the horizontal direction the relevant accumulation(s) are written to the statistics output buffer and the local RAM value is reset to zero.

The AE/AWB patch statistics output buffer is ordered from first to last patch data in the first row of patches in the lowest order enabled plane followed by first to last patch data in the first row of patches in the second lowest order enabled plane etc. up to the highest order enabled plane. This is then followed by second row of patches in the same order and so on until the last row of patches.

8.5.1.7 AF Statistics

Auto Focus (AF) statistics output may be enabled, independently from AE/AWB statistics. The patches are configured independently from the AE/AWB patches. They are configured in the same fashion, but with one restriction: the gaps between the patches, both horizontally and vertically, are hardcoded to zero. When operating on Bayer data, AF stats are accumulated on the Green channel only: red and blue pixels are ignored.

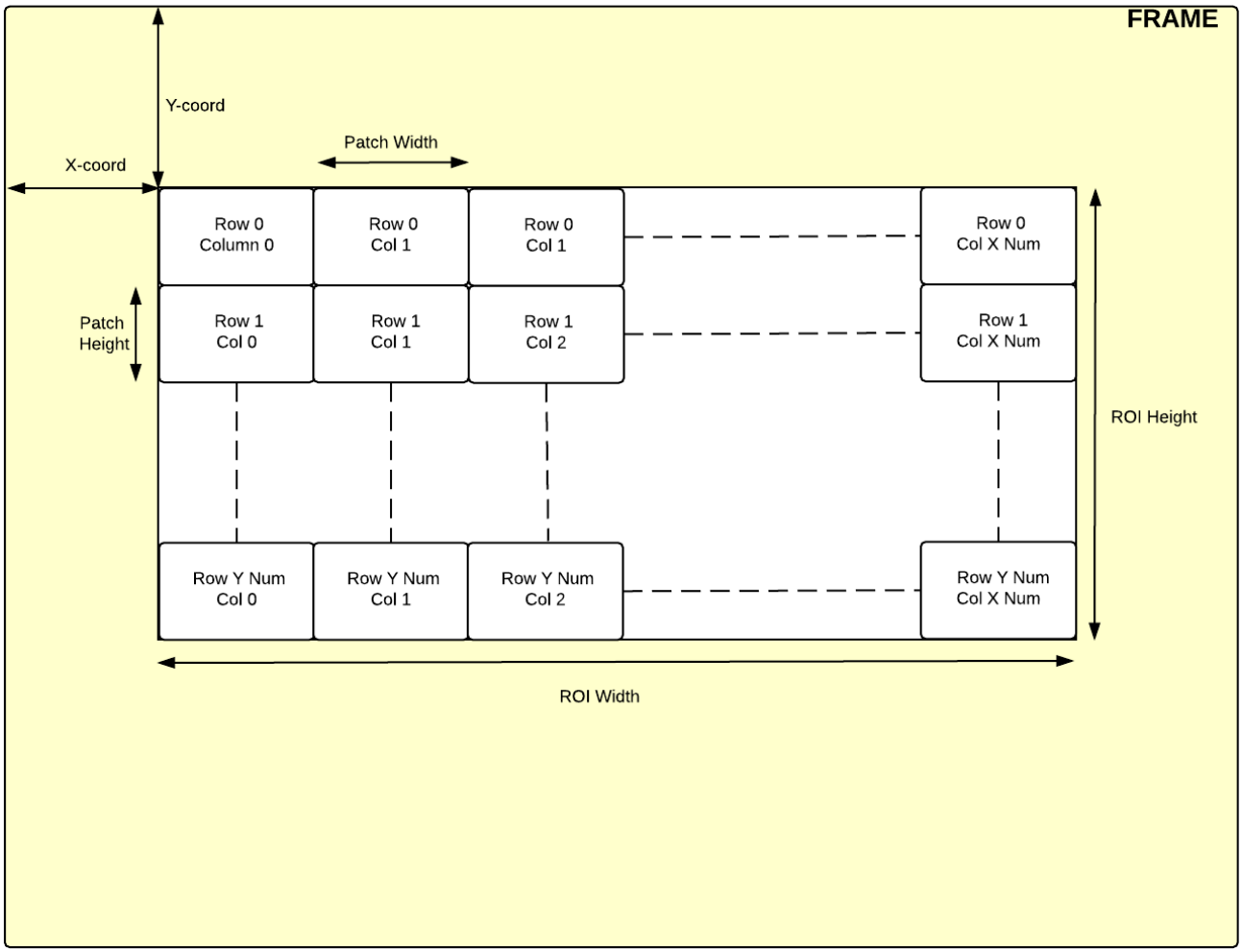


Figure 21: AF stats patch configuration, showing ROI (Region Of Interest)

First, we define the ROI (region of interest) as the bounding box of all AF patches. For each line, two separate IIR filters are run over every pixel within the ROI, from left to right. Each filter has a set of programmable coefficients, a programmable subtraction value, and a programmable threshold. The output of the two filters is then fed into per-patch accumulation logic. For each patch, the following values are output:

- Sum of all pixels output by filter 1 which are greater than a programmable threshold.
- Sum of all pixels output by filter 2 which are greater than a programmable threshold.
- Sum of the maximum values in each row from filter 1 output.
- Sum of the maximum values in each row from filter 2 output.
- Sum of all (unfiltered) input pixels in the patch.

8.5.1.8 Luma histogram

A histogram of the Luma channel is gathered across the frame on the designated plane, programmed via the `RawParam::statsPlanes` variable. For non-Bayer data this should correspond to the clear or white channel. For Bayer data a Luma channel is derived by convolving the input image with a 3x3 kernel with following coefficients:

```
[ [1, 2, 1],
  [2, 4, 2],
  [1, 2, 1] ] / 16
```

This kernel gives weights of R = 0.25, G = 0.5, B = 0.25 to each pixel, no matter what Bayer channel is at the center of the kernel. Note that the effect of this filter is to crop the image by one pixel on all sides such that the total number of pixels analyzed is reduced by (2*w + 2*h) – 4, where w and h are the width and height of the image respectively.

The histogram has 256 bins numbered 0 to 255, each containing a U32 count. Bins are indexed using the 8 most significant bits of each pixel. For example if the current pixel value is 0x369 and the configured bit-depth of the input RAW data is 11 bits then the 8 most `signstatsFrmDimificant` bits are 0x6d (decimal 109); the count in bin 109 is incremented by 1.

The histogram is stored in a local RAM during processing then uploaded to the statistics output buffer starting at bin 0 at the end of the frame. The next frame is stalled until the entire histogram has been uploaded. The count in each bin is reset to 0 at the start of each frame hence the histogram data uploaded to memory will only store data relevant to the designated plane of one complete frame.

8.5.1.9 RGB histogram

A histogram is computed independently for each of the R, G and B channels. In Bayer mode the histograms are computed over the same designated plane as the Luma histogram. In non-Bayer mode, the planes for the R, G and B histograms are specified via the corresponding fields in the `RawParam::statsPlanes` variable.

Each histogram has 128 bins numbered 0 to 127, each containing a U32 count. Bins are indexed using the 7 most significant bits of each pixel. For example if the current pixel value is 0x369 and the configured bit-depth of the input RAW data is 11 bits then the 7 most significant bits are 0x36 (decimal 54); the count in bin 54 is incremented by 1.

8.5.2 Configuration

The Raw filter is configured via the ***RawParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
FrmDim	31:16	Frame height
	15:0	Frame width
grgbPlat	29:16	Plato Bright – Maximum local green difference reduction in bright areas of the image.
	13:0	Plato Dark – Maximum local green difference reduction in dark areas of the image.
grgbDecay	31:16	Decay (slope) control of local green difference reduction in bright areas of the image. This value should be set >= the plato value for bright areas (grgbPlat[31:16]).
	15:0	Decay (slope) control of local green difference reduction in dark areas of the image. This value should be set >= the plato value for dark areas (grgbPlat[15:0]).
badPixCfg	31:16	Noise Level – This can be set to zero for images taken in good conditions, with low noise level. It should be set to correspond to the noise level (variance) to prevent noise from interfering with bad pixel detection.
	15:12	Alpha g hot – Filter aggressiveness control for Green Hot pixels.
	11:8	Alpha g cold – Filter aggressiveness control for Green Cold pixels.
	7:4	Alpha rb hot – Filter aggressiveness control for Red/Blue Hot pixels.
	3:0	Alpha rb cold – Filter aggressiveness control for Red/Blue Cold pixels.
cfg	27	Static defect correction enable. When set, a defect list is read from CMX memory via the filter's auxiliary input buffer's base address register. Static defect pixel correction is performed in accordance with the contents of the defect list.
	24	RGB histogram enable
	23:16	Bad pixel detection threshold
	13	AF stats output enable
	12	Gain Saturate mode: 0 – 4x4 mode 1 – 2x2 (Bayer CFA) mode
	11:8	Input data width – 1
	7	Luma histogram enable
	6	AE/AWB Stats output enable
	5	Hot/Cold green pixel only
	4	Hot/Cold pixel suppression enable
	3	Gr/Gb imbalance correction enable

Name	Bits	Description
	2:1	Bayer pattern. When Bayer input, same encoding as used for Bayer demosaicing configuration when non-Bayer data should be set to 0x00.
	0	RAW format: 0 – Planar data 1 – Bayer array data
gainSat[4]	[0] These values operate on GR pixels in bayer or 0, 4, 8, 12... etc in planar [1] These values operate on GR pixels in bayer or 0, 4, 8, 12... etc in planar [2] These values operate on GR pixels in bayer or 0, 4, 8, 12... etc in planar [3] These values operate on GR pixels in bayer or 0, 4, 8, 12... etc in planar	
	31:16	Level at which pixels will be clamped after corresponding gain is applied. Format is U16.
	15:0	Gain value applied to pixels. Format is U8.8.
statsBase	31:0	AE statistics buffer base – should be 8-byte aligned
statsPlanes		Active Planes for AE/AWB Stats collection. For non Bayer data 4 Patch Planes may be programmed. For Bayer data only the first patch plane will be active.
	21:20	The number of active patch planes -1. The Patch Planes used will be those programmed in the lower Patch Plane registers, i.e. if two patch planes are required then program active_patch_planes = 1 and program the required planes into Patch Plane 0 and Patch Plane 1.
	19:16	Luma Histogram Plane
	15:12	Patch Plane 3
	11:8	Patch Plane 2
	7:4	Patch Plane 1
	3:0	Patch Plane 0
statsFrmDim	31:16	Frame height
	15:0	Frame width
statsPatchCfg		Accumulation patch configuration
	31:24	Patch height – 1 (programmed as patch height minus 1)
	23:16	Patch width – 1 (programmed as patch width minus 1)
	13:8	Y_No – number of patches in vertically – 1
	5:0	X_No – number of patches in horizontally – 1
statsPatchStart		Start location of first (top-left) patch for AE/AWB statistics
	31:16	Y co-ordinate
	15:0	X co-ordinate

Name	Bits	Description
statsPatchSkip		Number of pixels from top-left of one cell to top-left of next for AE/AWB statistics.
	31:16	Value programmed should be Y skip -1.
	15:0	Value programmed should be X skip -1.
statsThresh	31:16	Dark pixel threshold. If a pixel (or any pixel in a 2x2 block, in the case of Bayer data) is less than this threshold, then the alternate accumulator set is used for that pixel (or 2x2 block of pixels). Threshold values have the same bit depth as the incoming data.
	15:0	Bright pixel threshold. If a pixel (or any pixel in a 2x2 block, in the case of Bayer data) is greater than this threshold, then the alternate accumulator set is used for that pixel (or 2x2 block of pixels). Threshold values have the same bit depth as the incoming data.
afF1coefs[11]		Coefficients for auto-focus stats filter 1.
	19:0	AF filter 1 coefficient N, in S(12,8) fixed-point format.
afF2coefs[11]		Coefficients for auto-focus stats filter 2.
	19:0	AF filter 2 coefficient N, in S(12,8) fixed-point format.
afMinThresh	31:16	Minimum threshold for per-patch accumulation of filter 2 output.
	15:0	Minimum threshold for per-patch accumulation of filter 1 output.
afSubtract	15:0	Value to be subtracted initially from pixels at beginning of IIR filtering.
afPatchCfg		Accumulation patch configuration for auto Focus Statistics.
		Patch height – 1 (programmed as patch height minus 1)
		Patch width – 1 (programmed as patch width minus 1)
		Y_No – number of patches in vertically – 1
		X_No – number of patches in horizontally – 1
afPatchStart		Start location of first (top-left) patch, and hence of the ROI, for auto-focus statistics.
	31:16	Y Coordinate.
	15:0	X Coordinate.
afStatsBase	31:0	Auto focus stats base address.
histLumaBase	31:0	Luma histogram base address.
histRgbBase	31:0	RGB histogram base address.

Table 15: RawParam structure user-specifiable fields

8.6 LSC filter

Input	Up to 14 bit RAW (Bayer pattern or non Bayer data) + sub-sampled gain mesh.
Operation	Lens shading/color shading correction via application of per pixel gains interpolated from sub-sampled gain mesh.
Filter kernel	Point operation based on bilinear interpolation of gain mesh (2x2 kernel). Maximum dimensions of gain mesh are 1023x1023. Gain mesh dimensions must not exceed dimensions of input image.
Local line buffer	No
Output	Up to 14 bit RAW (Bayer pattern or non Bayer data).
Instances	1

Lens shading correction (or anti-vignetting) compensates for the effect produced by camera optics whereby the light intensity of pixels reduces the further away from the center of the image they are. The compensation is applied by means of a gain map, generated during calibration, which provides a position-dependent correction. Color shading (color non-uniformity caused by CFA crosstalk) may also be corrected by this same operation when a separate gain map is available for each color channel.

A mesh based scheme is used for the gain map and bilinear interpolation is used to sample the mesh according to the image pixel location to generate the gain to be applied. The pixel is then simply multiplied by the gain to perform the correction.

The lens shading correction (LSC) filter can handle either Bayer pattern or planar data (e.g. RGBW from array sensor).

The bit depth of the input data may be between 6 and 16 bits. The input data width (in bytes) is programmable

8.6.1 Features

The input data may be in Bayer or Planar format. The mesh is provided as a separate input image, residing in CMX memory. The gain map data is stored in U8.8 format. The dimensions of the mesh are typically much smaller than the dimensions of the image. The hardware will scale the mesh to match the image size. The maximum mesh size is 1024x1024. The scaled mesh is simply multiplied by the input image in order to perform the correction. If the input image is in planar format, the mesh must also be in planar format. If the image to be corrected is in Bayer format, the 4 mesh planes must be interleaved into a Bayer mosaic pattern. The Bayer Order of the mesh must match the Bayer Order of the input image.

8.6.2 Configuration

This filter is configured via the ***LscParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
gmBase	31:0	CMX memory address of the gain correction mesh
gmWidth	9:0	Width of the gain mesh – must be a multiple of 4

Name	Bits	Description
gmHeight	9:0	Height of the gain mesh
dataFormat	0	0 = Planar, 1 = Bayer
dataWidth	3:0	Bits per pixel of the input data. Valid values are in the range [6, 16]. If the specified value is 8 or less, the data for each pixel is packed into a single byte. If the specified value is more than 8, the data for each pixel is packed into two bytes.

8.7 Debayer / demosaic filter

Input	RAW Bayer data in up to 14 bits
Operation	Configurable in either high quality or preview mode. High quality: local luma adaptive hybrid Bayer demosaicing filter combining an optimized implementation of AHD and bi-linear interpolation. Preview: downsize demosaic; optimized demosaic and downsize by two in both dimensions.
Filter kernel	High quality: 5x11 Preview: 4x4
Local line buffer	Yes, maximum supported image width is 4624
Output	Planar RGB in up to 16 bits (sub-sampled in preview mode) FP16/U8F Luma data
Instances	1

The Bayer demosaicing filter supports high-quality demosaicing, in addition to preview image generation. The output data may be RGB and/or Luma.

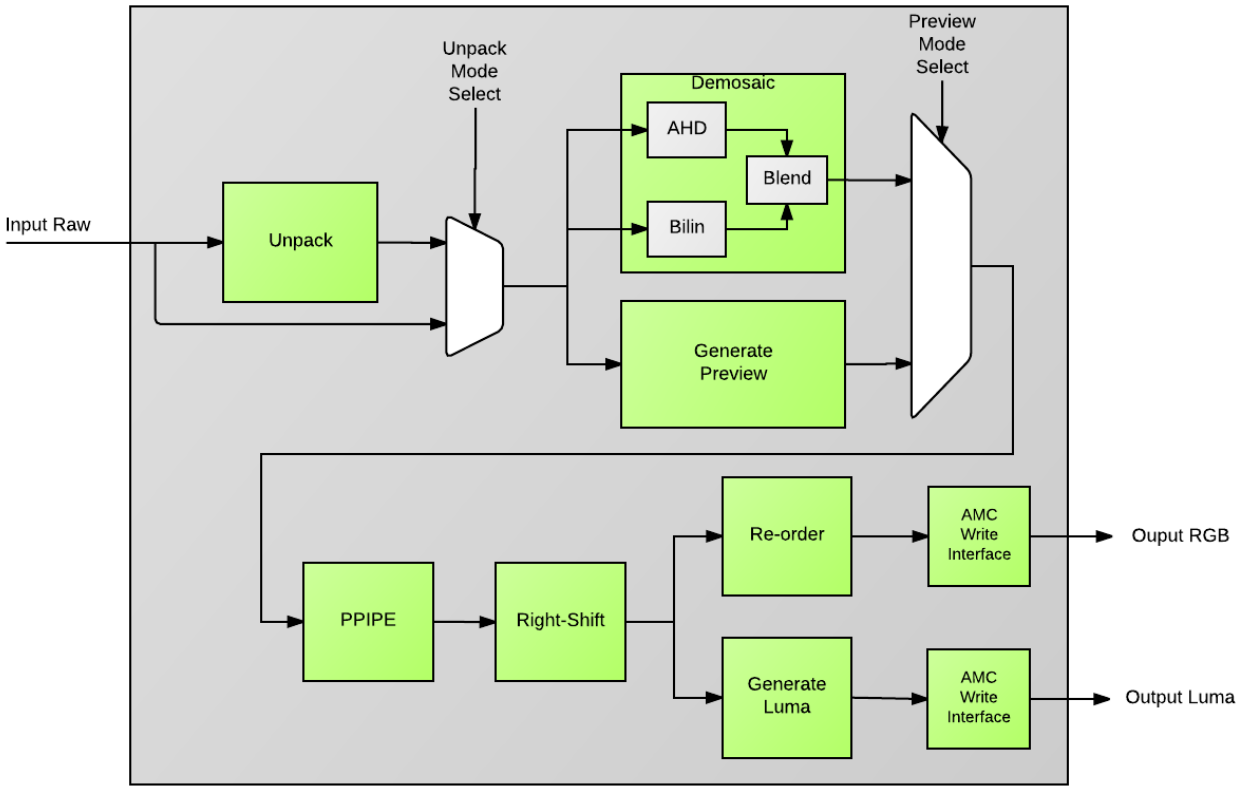


Figure 22: Top-level overview of Bayer Demosaicing filter

8.7.1 Features

8.7.1.1 Packed input

The Bayer-domain input to the block may be either packed or unpacked. Unpacked input is either 8 or 16 bits per pixel. If the number of input bits is either 10 or 12, then packed input is supported. The supported packed formats match the packed formats supported by the RAW filter, as described in section 8.5.

8.7.1.2 Demosaicing

Bayer color filters are used in most single chip digital image sensors. Each pixel is filtered to record only one of 3 color components: red, green or blue. Bayer *demosaicing* algorithms work on a window (or kernel) to interpolate, from the neighboring pixels, a set of complete RGB (red, green and blue) values for each pixel.

The SIPP Bayer demosaicing filter uses a hybrid approach. A highly optimized implementation of the AHD (Adaptive Homogeneity Directed) algorithm is run in parallel with a simple bi-linear interpolation. The AHD algorithm is a much more sophisticated algorithm, but can produce undesirable artefacts in dark areas of the image, where the SNR is low. The bilinear filter performs classic bi-linear interpolation, but with zipper artifact avoidance. The filter runs a local luma approximation and merges the output of the AHD algorithm with the bi-linear interpolation output according to the luma intensity. The merging of AHD and Blinear are controlled via a programmable slope and offset.

All 4 RGB Bayer patterns are supported. The pattern is defined by the pixel colors of the first two pixels on the first two lines of the Bayer image: e.g. if the first line contains red (R) and green and starts with a green (G) pixel then the second line will contain blue (B) and green and start with a blue pixel; the Bayer pattern is GRBG; if the first pixel is red then the Bayer pattern is denoted as RGGB. The appropriate Bayer pattern should be programmed in the DbyrParam::cfg variable; the 4 Bayer patterns and their encoding are as follows:

- GRBG = 0
- RGGB = 1
- GBRG =2
- BGGR = 3

(for support of RGBW #1 the GRBG pattern is programmed this is so that the position of the White pixels is in the green location of the Bayer pattern).

8.7.1.3 Up Conversion of input data

The filter is implemented in 16 bit arithmetic, so to make the most use of the available range input data of fewer than 16 bits is promoted to the 16 bit range by left-shifting and filling the lower order bits by replication of the MSBs of the input data as shown below.

E.g. conversion of 8 bit to 16 bit as follows:

```
RAW16 = RAW8 << 8 | RAW8
```

E.g. conversion of 10 bit to 16 bit as follows:

```
RAW16 = RAW10 << 6 | RAW10 >> 4
```

The width of the input data is specified in the DbyrParam::cfg variable. If the width of the output data is specified as fewer than 16 bits it is taken from the MSBs of the 16 bit RGB results.

8.7.1.4 Preview Mode

As an alternative to the full-resolution high-quality demosaiced output, a preview version of the image may be output instead. The preview image is also RGB, but is one half of the input resolution in each dimension. The preview image is generated by a "smart binning" filter, which reduces the resolution and converts from Bayer to RGB in one step. Preview mode is intended to reduce power consumption when full-resolution output is not needed, since the remainder of the ISP pipeline only needs to process ¼ the number of pixels. When in preview mode, this filter, and the ones in the pipeline that follow it, only need to be invoked half as often – this filter only needs to be invoked once for every 2 lines produced by its parent filter.

8.7.1.5 Luma Output

A Luma version of the image may also be output. Both Luma output and RGB output may be independently enabled/disabled. The Luma is generated from RGB using programmable coefficients.

"Worms" are an artefact that appear where the signal-to-noise ratio is low, i.e. in dark area of the image, or in low-light conditions. The demosaic block performs two types of demosaicing: one suitable for areas where SNR is low (this type of demosaicing is not susceptible to worms), and one suitable for more normal conditions. The filter adaptively combines the two outputs, according to some programmable controls. The output suitable for low-SNR is preferred where the local luminance is lower, and the output for high-SNR is preferred where the local luminance is higher, and also in the neighborhood of strong edges (high local gradient).

The Low-SNR and High-SNR output for the Red channel are combined as follows:

$$alpha = \left(luma + g \cdot gradient_{m}ult + offset\right) \cdot slope \quad (1)$$

$$Rout = Rlow \cdot \left(1 - alpha\right) + Rhigh \cdot alpha \quad (2)$$

Where *Rlow* is the red channel of the demosaiced output for low-SNR conditions, *Rhigh* is the red channel of the demosaiced output for normal conditions, *luma* is the local brightness, and *g* is an estimate of the local gradient. *Slope*, *offset*, and *gradient_mult* are user-programmable controls. *alpha* is scaled (divided by 2^bit_depth-1) and clamped within the range [0, 1] after it is calculated. The Green and Blue channels are processed in the same way as the Red channel.

8.7.2 Configuration

This filter is configured via the ***DbyrParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	31:24	Gradient multiplier **gm** in fixed-point U(1,7) format used to moderate fade to bilinear interpolation in the presence of strong horizontal/vertical gradients
	16:15	Plane multiple **pm** for processing multiple planes of Bayer data. Default value is for 1 input Bayer plane, therefore 3 output planes (RGB). Value programmed is value minus 1
	14:12	Image Order Out 000 – RGB 001 – BGR 010 – RBG

Name	Bits	Description
		011 – BRG 100 – GRB 101 – GBR
	11:8	Output data width – 1
	7:4	Input data width – 1
	3	Force RB to zero. For use with RGBW data
	2	Luma only Hmap. For use with RGBW data
	1:0	Input Bayer pattern: 00 – GRBG 01 – RGGB 10 – GBRG 11 – BGGR (for RGBW data set this register to 00 - GRBG)
thresh		Color Moire avoidance thresholds, preview mode, Luma and RGB generation enables
	28	Enable Preview: 0 – AHD/Bilinear demosaic mode is enabled, and the preview generation block is bypassed. The output resolution is the same as the input resolution. 1 – Preview mode is enabled, and the Demosaic block is bypassed. The output resolution is one half of the input resolution in each dimension.
	27	Enable Luma write client
	26	Enable Luma generation and Luma streaming output
	25	Enable RGB output
	24:13	Threshold for b
	12:0	Threshold for a
dewormCfg	31:16	De-worming offset (*offset* in equation 1 above). In unsigned 8.8 fixed point format.
	15:0	De-worming slope (*slope* in equation 1 above). 16-bit signed integer.
lumaWeight	23:16	Weight applied to Red channel for Luma generation
	15:8	Weight applied to Green channel for Luma generation
	7:0	Weight applied to Blue channel for Luma generation

8.8 DoG / LTM filter

Input	FP16/U8F Luma data, 2 read clients are used (older version of data is read by the second instance, to match internal latency of filter)
Operation	Local Tone Mapping plus Noise reduction based on a Difference of Gaussians
Filter kernel	Up to 15x15
Local line buffer	Yes, maximum supported image width is 4624
Output	Up to 16 planes of FP16/U8F
Instances	1

The DoG/LTM (Local Tone Mapping) filter is designed to operate on the Luma path of an ISP pipeline, and performs two major functions:

- Removal of Low Frequency noise via a Difference of Gaussians (DoG) technique
- Application of Local Tone Mapping (LTM)

8.8.1 Features

8.8.1.1 DoG Based Denoise

The DoG denoise filter is designed to remove low-frequency noise. Since the kernel size can be up to 15x15, it is capable of removing lower-frequency noise that the primary 7x7 Luma denoise filter is not able to remove. This low-frequency noise is normally only a problem in poor lighting conditions. The idea is to isolate frequencies within the image which are within a given band. This is done by subtracting two Gaussian-filtered versions of the image. The programmer is responsible for calculating the kernel coefficients. There are two sets of coefficients, one for each Gaussian kernel. When generating the coefficients, values of Sigma should be chosen such that the appropriate band of frequencies are isolated. After calculating the Difference of Gaussians, values whose absolute value are greater than a programmable threshold are set to zero. The thresholded bandpass image is then subtracted from the original image, after being multiplied by a programmable gain in the range [0, 1.0]. This has the effect of removing the isolated frequencies from the original image. However, due to the thresholding operation, strong edges are not suppressed. The filter is very effective at removing low-frequency noise in flat areas. It is designed to be used in conjunction with the Luma Denoise filter, and should be applied prior to Luma Denoise.

The programmable kernels must be symmetrical and separable. For each kernel, a single one-dimensional kernel is programmed, and this kernel is applied first in the vertical direction, and then in the horizontal direction. One kernel has a size of 15x15, and the other has a size of 11x11. The 15x15 Gaussian kernel is expected to generated using a larger value of Sigma. The output of the 15x15 kernel is subtracted from the 11x1 kernel to compute the difference of gaussians. Since the kernels are symmetrical, only 6 coefficients are programmed for the 11x11 kernel (the other five are automatically generated internally by mirroring) and only 8 coefficients are programmed for the 15x15 kernel.

The DoG filter operates in two modes:

- Denoise mode.
- DoG-only mode.

In DoG-only mode, the filter outputs the Difference of Gaussians directly. The following restrictions apply

when the DoG filter is in DoG-only mode:

- LTM is implicitly disabled.
- Only FP16 is supported as the output format (U8F output is not supported).
- The line width is limited to 2312.

In denoise mode, DoG and LTM may be independently bypassed/disabled.

NOTE: When operating in Denoise mode, the data being input to the Gaussian filters will have been subsampled spatially by simple averaging of each 2x2 block, before being stored in an internal line buffer store. Prior to applying the Gaussian kernels, the data is upsampled back to the original resolution, using bilinear interpolation. Since this subsampling and subsequent resampling removes some of the higher frequencies from the image, the net effect of the filter is different than if the Gaussian kernels were run directly at the full resolution, and therefore the Gaussian kernels that are specified need to be modified slightly (different values of Sigma) to achieve the same effect.

When operating in DoG-only mode, the subsampling and subsequent upsampling steps do not take place.

8.8.1.2 Local Tone Mapping (LTM)

The local tone mapping filter is used to apply a spatially-varying contrast curve to the Luma channel, to bring out details in the image. The idea is to have the contrast curve at its steepest around the range of pixel intensities that are to be enhanced (given maximum contrast). Take the following examples of contrast curves:

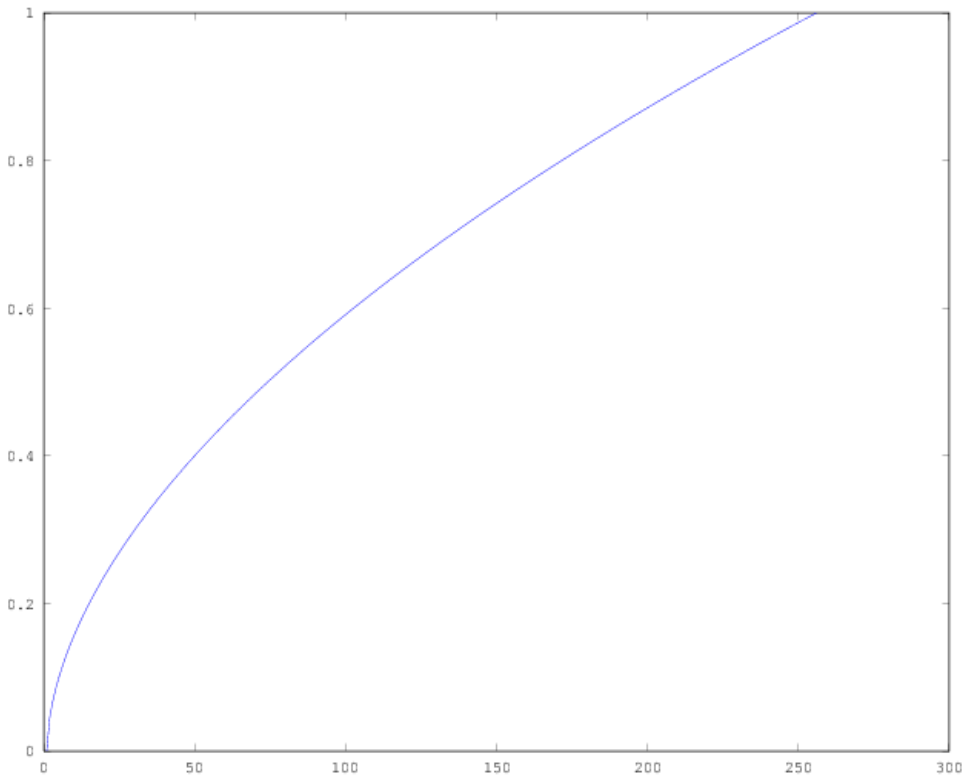


Figure 23: A gamma curve of 1/1.8 increases contrast in the shadows, but compresses the highlights

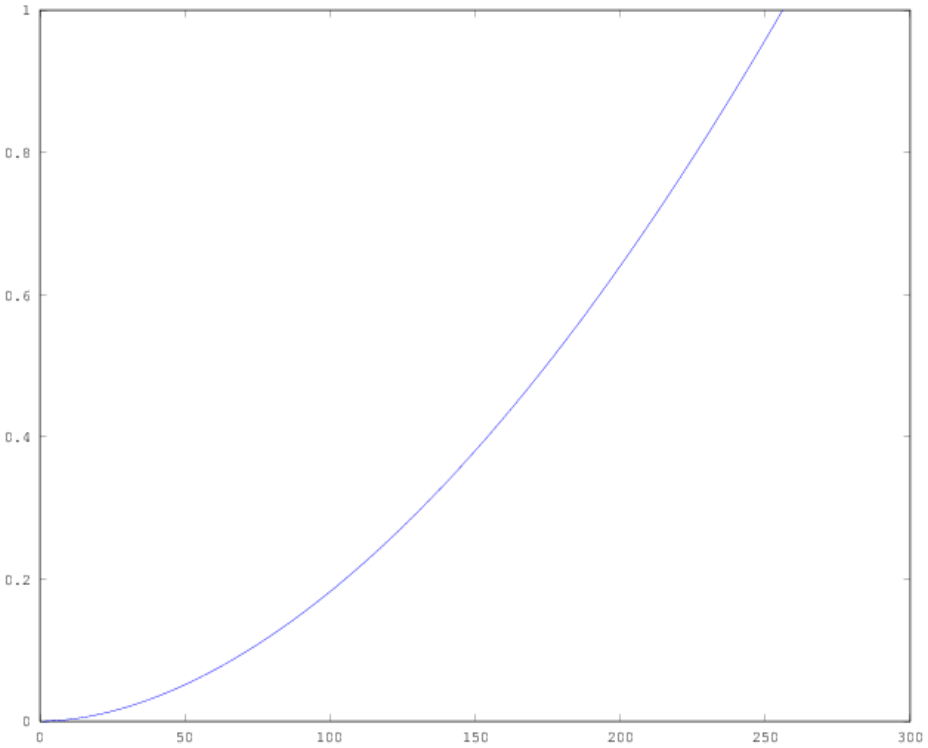


Figure 24: A gamma curve of 1.8 increases contrast in the highlights, but compresses the shadows

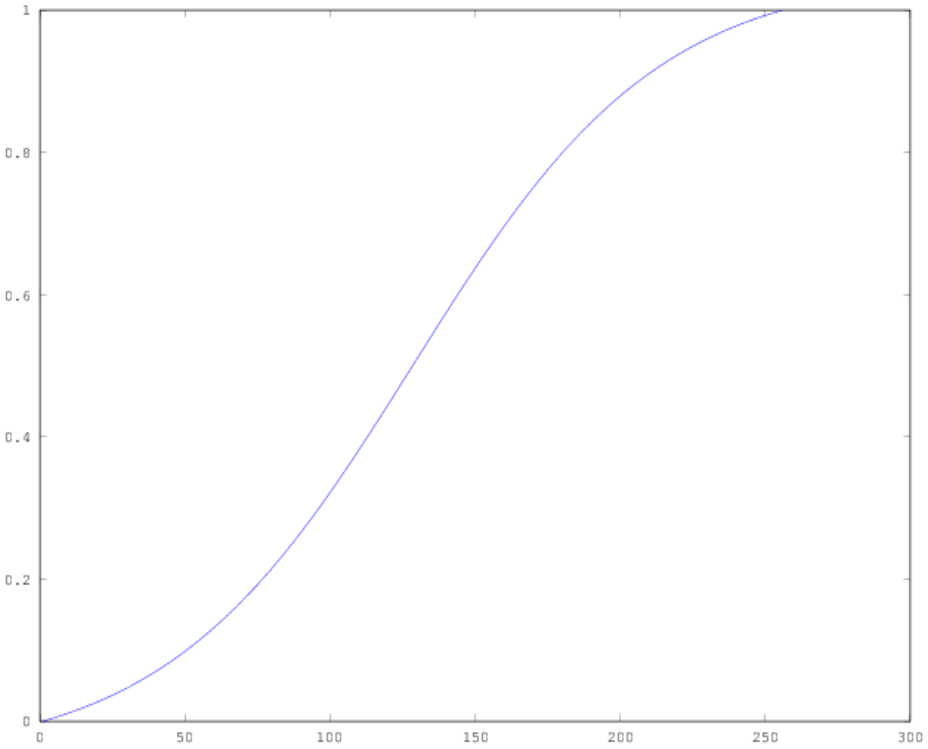


Figure 25: An S-curve or sigmoid increases contrast in the mid-tones, but compresses the shadows and highlights

If any of the above curves are applied globally to the Luma channel, then the contrast will be improved in some areas of the image, but at the expense of decreasing contrast in other areas. What we need is a spatially adaptive tone mapping operator, which stretches the highlights in bright areas (e.g. enhancing the appearance of clouds in the sky), while also stretching the shadows in dark areas. This filter is such a filter. A pre-generated set of tone curves are supplied by the user, and the appropriate curve is selected and applied based on the light level of the surrounding area, or locality.

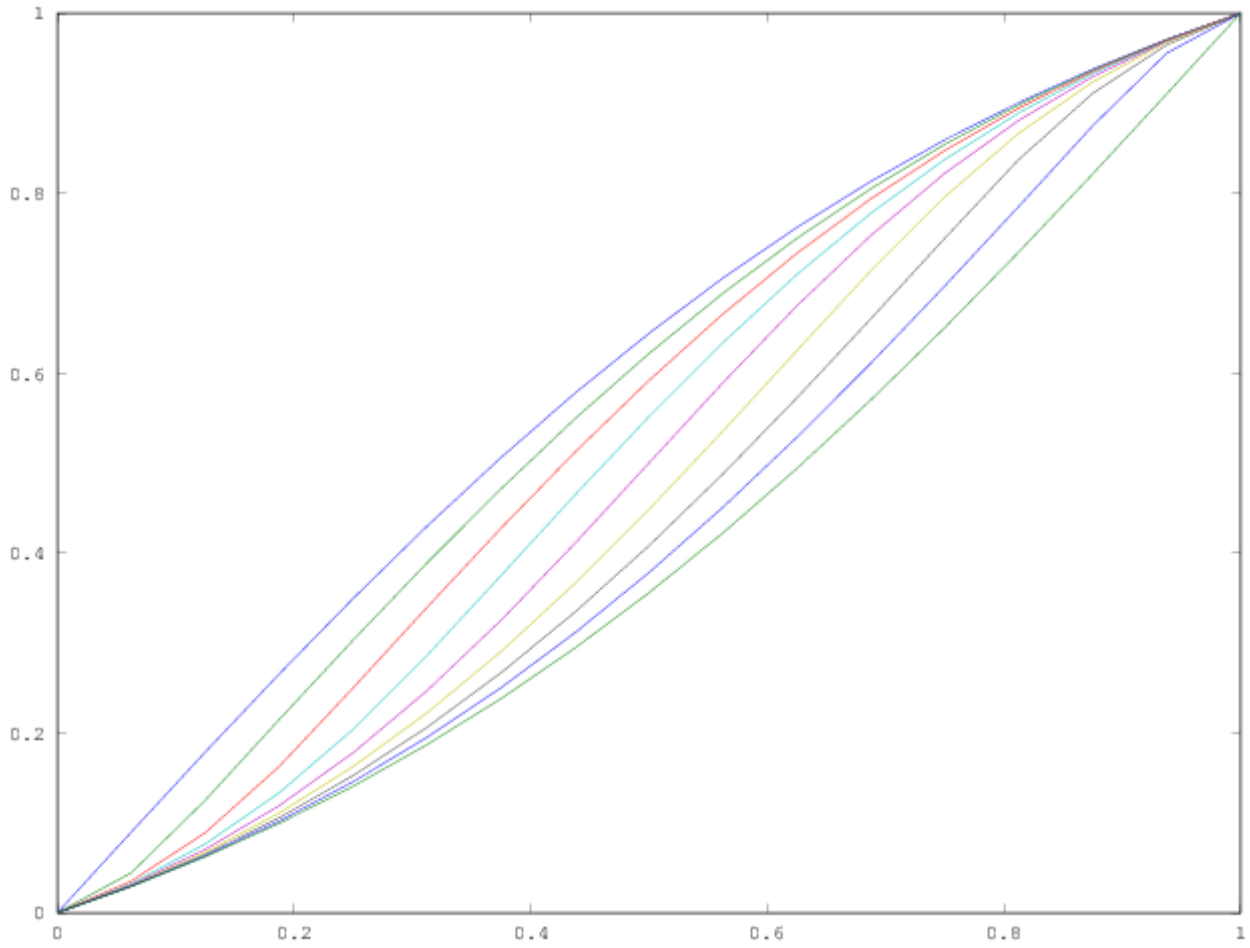


Figure 26: A set of curves generated for use by the LTM filter

Above is a set of curves which could be used by the filter. Different curves have maximum contrast at different intensity levels. The filter is programmed with 8 curves, each curve having 16 knee points. The filter selects two curves based on the local (or background) intensity, then interpolates between two knee-points along the selected curves. It is, essentially, a bilinear filter, which samples an 8x16 “image”.

The curves are stored as a 2D array of U(12,0) values, with each column storing a single curve. Bilinear interpolation is performed based on two co-ordinates: on the X axis, the local background intensity is used to select the curves, and on the Y axis, the incoming pixel value is used to select the knee points:

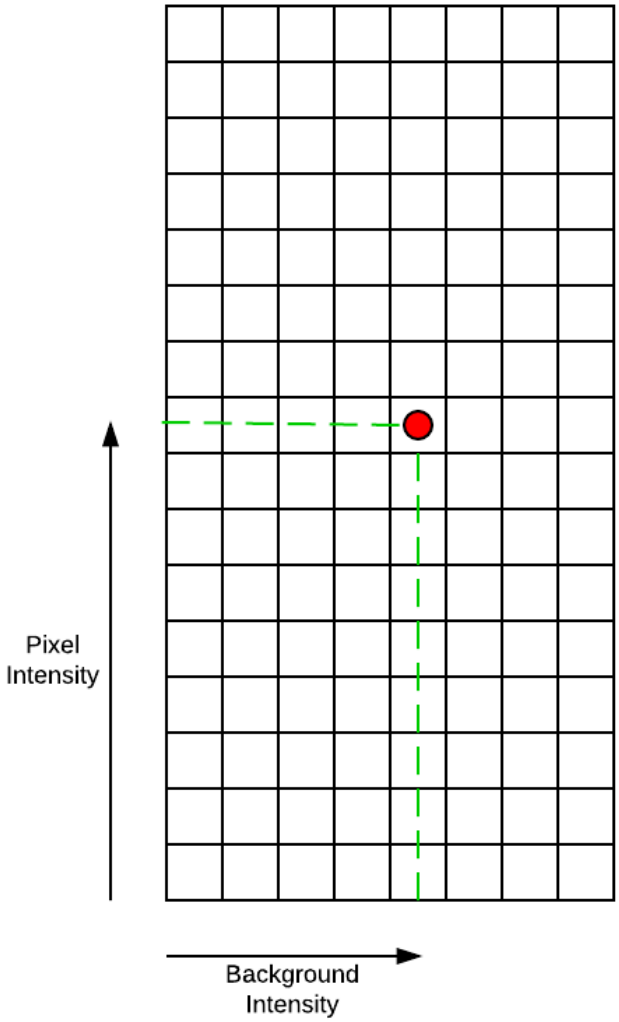


Figure 27: Bilinear interpolation: curve selection based on background intensity, and knee-point selected based on single pixel intensity

The local background intensity is calculated using an 11x11 filter. This filter approximates a bilateral filter, and heavily blurs the image, without blurring across strong edges.

To use the Local Tone Mapping block, generate the 9 curves to be applied at the various image intensity levels. Note that the first curve corresponds to a background intensity of 0, and the final curve corresponds to a background intensity of 1.0. Note also that each curve typically has a value of 0 at the first knee-point, and a value of 1.0 at the final knee-point. By setting the first and last knee points of each curve to 0 and 1.0 respectively, and setting the rest of the knee-points in each curve to lie along a straight line, the filter is essentially bypassed.

There is also a programmable threshold to control the Background Intensity generation filter. This threshold controls the strength of edges that the filter will not cross. If set to its maximum value, the filter acts as an 11x11 box filter.

Note that the filter internally subsamples the input to the Background Intensity filter by simple averaging of each 2x2 block. The Background Intensity filter runs directly on the subsampled data, so the filter has an effective support of 22x22 pixels in terms of the full image resolution. The output of the Background Intensity filter is then upsampled to the full image resolution using bilinear interpolation before it is input to

the bilinear sampling stage.

8.8.2 Configuration

This filter is configured via the ***DogLtmParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	29:26	Valid values are 3, 5, 7, 9, 11, 13 and 15. Limits the support of the DoG Gaussian filters in the vertical direction. Only the specified number of lines are fetched from the line buffer, and lines outside of the fetched lines are implicitly filled with zeros, which means that the corresponding filter coefficients are ignored during the vertical filter pass. The central lines are always fetched, such that the center line's values are multiplied by the center coefficient regardless of this register setting.
	25:22	Number of planes used in LTM upsample.
	21:14	Threshold for LTM Background Generation filter.
	13:12	Local line buffer downscale rounding mode: 00 – Propagate carry bit. 01 – Don't propagate carry bit. 10 – Horizontal average uses fixed carry-in of 1, vertical average uses fixed carry-in of 0.
	10	Set to 1 to clamp the filter FP16 output into the range [0, 1.0].
	9:2	U8F threshold. If the absolute value of the Difference of Gaussians is greater than this threshold, it it set to zero.
	1:0	Operational mode: 00 – DoG only mode (LTM is implicitly bypassed): output is the thresholded Difference of Gaussians. 01 – LTM-only mode, DoG is bypassed/disabled. Output is the local tone mapped input. 10 – Denoise mode, LTM is bypassed/disabled. Output is the input minus the thresholded Difference of Gaussians. 11 – DoG+LTM mode: output is local tone mapped input minus the thresholded Difference of Gaussians.
dogCoeffs11	31:0	Pointer to Coefficient 0 → 5 of the 11x11 kernel.
dogCoeffs15	31:0	Pointer to Coefficient 0 → 7 of the 15x15 kernel.
dogStrength	7:0	Gain applied to difference of gaussians. Format is U8F.
ltmCurves	31:0	Pointer to table of curve table entries in U12F format.

8.9 Luma Denoise Filter

Input	FP16/U8F Luma data.
Operation	Luma denoise using an advanced weighted averaging algorithm.
Filter kernel	11x11
Local line buffer	Yes, maximum supported image width is 4624.
Output	FP16/U8F Luma data
Instances	1

The Luma Denoise filter applies a 7x7 filter to the Luma channel using advanced weighted averaging. The weights are calculated on a reference image, which is generated internally by the filter from the input. The reference image is generated in a way that gives control to the user over the filter strength as a function of the image intensity, and also as a function of the distance from the center of the image. In the diagram below, this reference image is generated by the "Gen Denoise Ref." block.

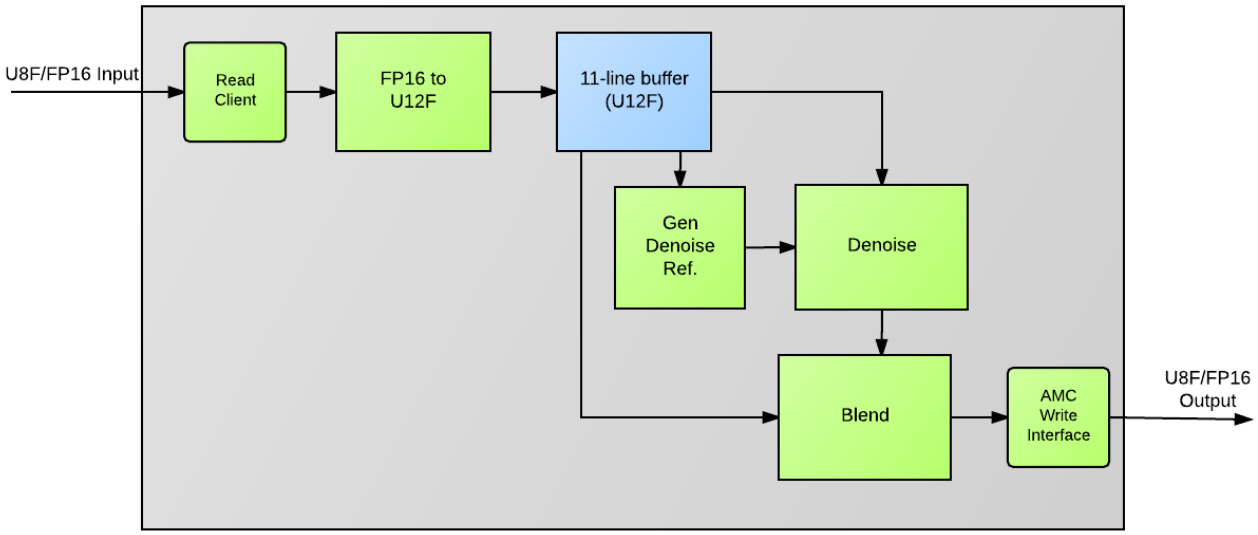


Figure 28: Top-level data flow of Luma Denoise filter

The reference image is used for weight calculation only. Once a weight has been calculated for each pixel in the 7x7 neighborhood, the output is calculated as a weighted average of the input pixels as follows:

$$I_{out} = \frac{\sum_{i=1}^{n} W_i I_i}{\sum_{i=1}^{n} W_i}$$

where I_i are the input pixels in the 7x7 neighborhood, and W_i are the calculated weights at the corresponding positions.

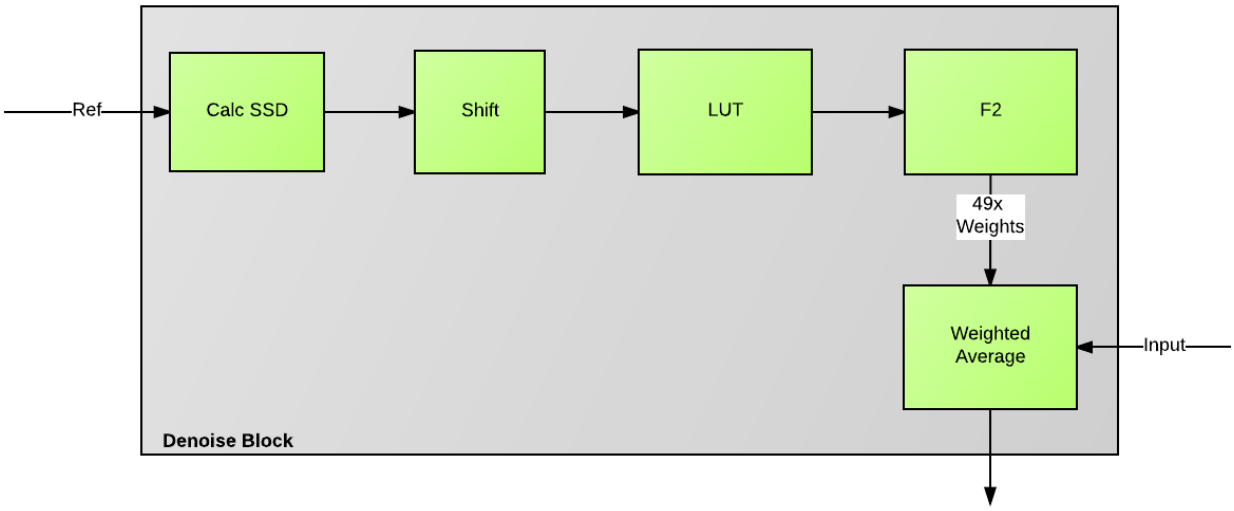


Figure 29: Denoise sub-block

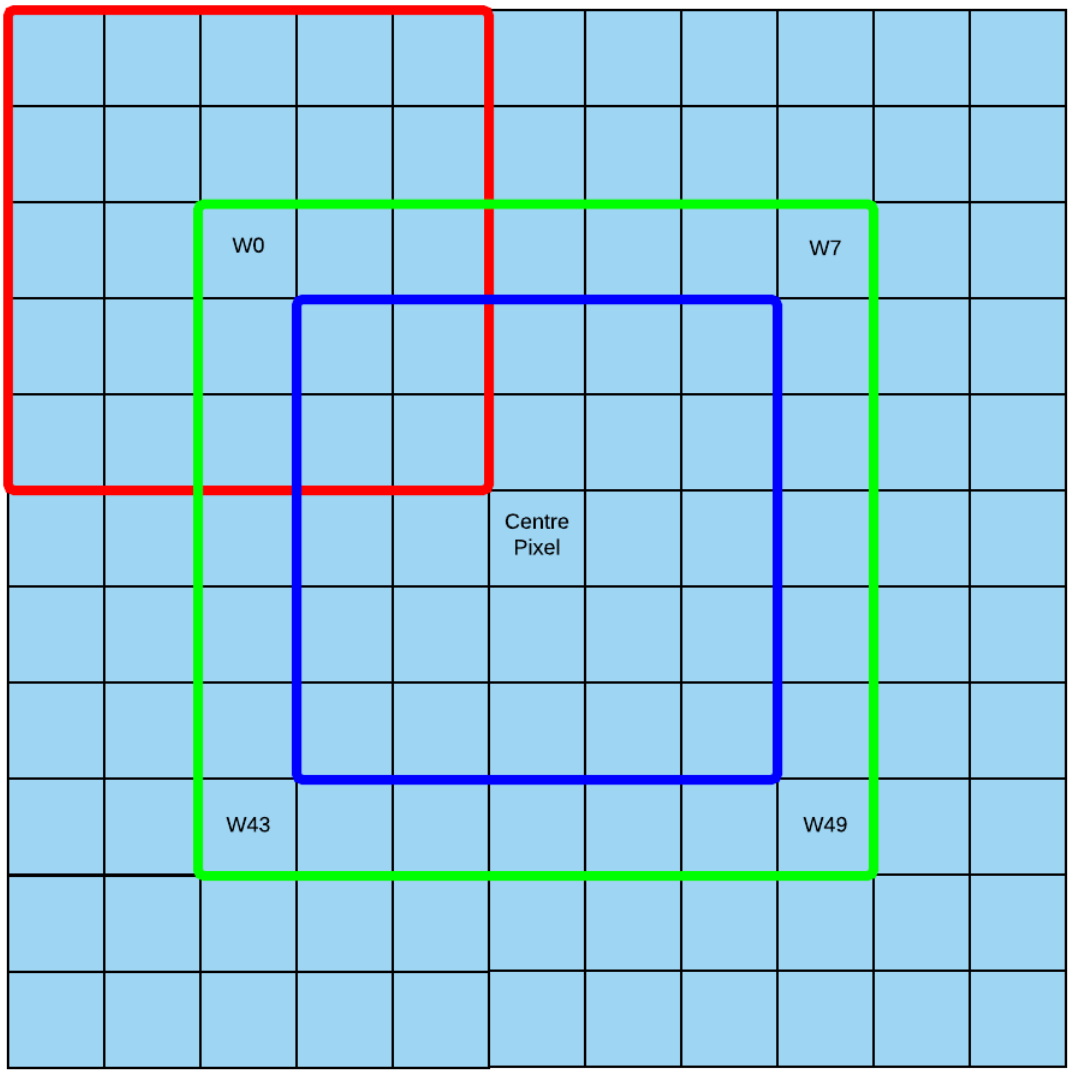


Figure 30: SSD windows for weight computation

Each weight is calculated based an Approximated Sum of Squared Differences (ASSD) value calculated by comparing a 5x5 window around the center pixel to a 5x5 window around the neighborhood pixel. In the diagram above, the green box highlights the 7x7 area, within which a weight must be calculated for each neighborhood pixel. The blue box shows a 5x5 window around the center pixel, which is used for each ASSD calculation. The red box shows the 5x5 window of pixels which is used for calculating weight W0. Effectively, 49 such 5x5 ASSD operations are needed to compute all of the weights for a single output pixel (the implementation is optimized to minimize the computation required). Since 7x7 weights are needed, and the ASSD window is 5x5, 11 lines of input data need to be present in the filter's local line buffer.

Each ASSD result is shifted, then mapped through a LUT operation. The shift amount and the LUT contents are programmable, and are chosen so as to control the strength of the filter (see below). Finally, the resulting weights are modified by the “F2” kernel, before the weighted averaging is performed.

The F2 kernel allows each weight to be multiplied by a factor of 1, 2, 4 or 8. This allows each weight to be modified as a function of spatial distance from the center pixel. The F2 kernel must be symmetrical. Therefore, only the upper-left 4x4 portion of the kernel is programmable, with the hardware generating the rest by mirroring.

C0	C1	C2	C3	C2	C1	C0
C4	C5	C6	C7	C6	C5	C4
C8	C9	C10	C11	C10	C9	C8
C12	C13	C14	C15	C14	C13	C12
C8	C9	C10	C11	C10	C9	C8
C4	C5	C6	C7	C6	C5	C4
C0	C1	C2	C3	C2	C1	C0

Figure 31: F2 kernel co-efficients

The 16 F2 coefficients are specified via a single 32-bit register. The register via which the F2 coefficients are programmed has 2 bits per co-efficient. The 2-bit codes translate into weight multipliers as follows:

Bitfield contents	Multiplier
0	1
1	2
2	4

Bitfield contents	Multiplier
3	8

After the weighted averaging, the resulting denoised pixel value is blended with the original, non-denoised input. The blending function is controlled by a programmable value, alpha, which is in the range [0, 1]. Specifying a large value of alpha gives more weight to the denoised pixel value, whereas a smaller value mixes more of the original input with the output. This allows a tradeoff to be made between aggressive noise removal, and detail preservation. When tuning the filter parameters, the denoise filter strength (which is controlled by the bitshift and LUT parameters described below) must be set high enough to avoid residual noise artifacts, or "noise islands", and is thus constrained to a certain extent. The "alpha" parameter gives additional control over noise reduction strength versus detail preservation.

8.9.1 Features

8.9.1.1 Generating the filter parameters

8.9.1.1.1 Weight formula LUT and bit position

The following C code shows how the computed ASSD values are converted to weights by the hardware:

```
assd = assd >> bitpos;
if (assd > 31)
      weight = 0;
else
      weight = lut[assd];
```

The following Matlab code show an example of how to compute the "lut" and "bitpos" programmable parameters:

```
% Generate programmable parameters based on desired denoising strength
function [ lut bitpos] = makelut(strength)
    if strength < .001
        % Avoid division by 0
        strength = .001;
    end
    if strength > 2047
        % Limit to prevent 'bitpos' > 11
        strength = 2047;
    end
    bitpos = floor(log2(strength)); % MSB position
    if bitpos < 0
        bitpos = 0;
    end
    npot = 2.^bitpos;       % nearest power of two (rounding down)
    alpha = (strength - npot) / npot;
    divisor = 4*(1-alpha) + 8*(alpha);
    sigma = .05;
    if strength < 1
        % Reduce sigma when 0 < strength < 1
        sigma = sigma * strength;
```

```
        end
        lut = (0:31)/31/divisor;
        lut = exp(-(lut.^2) / (2*(sigma.^2)));  % Gaussian

        % Quantize the LUT entries to 16 distinct values
        lut = int32(uint8(lut*16-1));
    end
```

The input to the above function is a simple “strength” parameter, and the function is designed so that the effectiveness of the filter increases monotonically as this “strength” parameter increases. In this example, the LUT is populated with an approximation of a Gaussian curve. As “strength” increases, the shape of the curve changes from a narrow Gaussian to a wider one, which leads to larger output weights. However, when a threshold is crossed (“strength” advances to the next power of two), we increment “bitpos” and transition back to a narrower Gaussian. Recall that if (assd>>bitpos) is greater than 31, the LUT is bypassed, and the weight is set to 0. Otherwise, the 5 bits of (assd>>bitpos) are used to index the LUT. By making “bitpos” larger, we select the 5 bits from a higher range of assd, thereby allowing larger values of assd to produce non-zero weights. Larger values of assd occur when the area surrounding the center pixel is dissimilar from the area surrounding the neighborhood pixel. Larger values of “bitpos” result in larger output weights and therefore more aggressive averaging of the center pixel with its neighbors.

8.9.1.1.2 Distance-based LUT

The distance-based LUT allows the user to control the strength of the denoise, as a function of distance from the center of the image. Since the image being denoised usually already has anti-vignetting applied, and since the SNR decreases further away from the center of the image (since less light reaches the corners of the sensor compared to the center), it is desirable to attenuate the reference image values as a function of distance from the image center. The stronger the attenuation, the stronger will be the effect of the denoise filter. The attenuation is normally calculated using the cosine-fourth law, which models the lighting falloff as a function of distance from the image center. The goal is to modify the pixel intensity, I, according to the following formula:

$$I = I * \cos\left(\sqrt{\left(x^2 + y^2\right)}\right)^4$$

The hardware internally computes $x^2 + y^2$. The result is then right-shifted by a programmable amount, to bring it into the range [0,255]. The in-range value is then mapped through the LUT, which performs the square-root and cosine-fourth operations in one step. Since the LUT is generated by software, the user is not restricted to using the cosine-fourth model of lens shading falloff.

The following Matlab code shows an example of generating the programmable bitshift parameter and the LUT entries for a given image size. The “angle” parameter controls the strength of the lighting falloff, which is a function of the Angle Of View of the sensor. The specified value is half of the diagonal angle of view, in radians.

```
function [ lut bitshift ] = gen_lut_bitshift(angle, width, height)
    w2 = width/2;
    h2 = height/2;

    % Maximum value that can be put into LUT
    maxval = w2^2 + h2^2;

    % Find how many bits to shift values right by, so that we can use
    % an 8-bit LUT
```

```
    shift = floor(log2(maxval)) + 1 - 8;

    x = 0:255;
    x2_plus_y2 = bitshift(x, shift);
    cornerval = sqrt(w2^2+h2^2);
    lut = round((cos(sqrt(x2_plus_y2) / cornerval * angle) .^ 4) * 255);
end
```

8.9.1.1.3 Intensity curve

To give the user control over the strength of the denoise as a function of pixel intensity, a gamma-like curve can be applied to the reference image. The shape of the gamma curve is controlled by two 9-element user-progammable LUTs. The first LUT is applied to pixels in the range [0,31], and the second LUT is applied to pixels in the range [32,255]. Since the curve is steepest for darker pixels, a more fine-grained LUT is needed for the lower part of the [0,255] range.

The following Matlab code shows an example of generating the LUT entries for a given Gamma curve:

```
% Generate two 9-entry LUTS: one which spans the range [0,32], and one
% which spans the range [32, 255]
x = 0:255;
lut1 = uint8((x(round(linspace(1,33,9)))/255) .^ gamma * 255);
lut2 = uint8((x(round(linspace(1,256,9)))/255) .^ gamma * 255);
```

8.9.2 Configuration

This filter is configured via the ***YDnsParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	24	Clear and set to '1' to enable update of the Cosine 4th law look-up table used to generate the reference image
	20:16	Amount to right-shift the result of x^2+y^2 before applying the distance-based LUT when generating the denoise reference image
	15:8	Output weight
	3:0	Bit position, controls right-shift of differences prior to indexing LUT
gaussLut[4]	32 4-bit LUT entries where gaussLut[x] is	
	31:28	LUT entry (x*8) + 7
	27:24	LUT entry (x*8) + 6
	23:20	LUT entry (x*8) + 5
	19:16	LUT entry (x*8) + 4
	15:12	LUT entry (x*8) + 3
	11:8	LUT entry (x*8) + 2
	7:4	LUT entry (x*8) + 1
	3:0	LUT entry (x*8) + 0

Name	Bits	Description
f2	F2 4x4 2-bit LUT entries	
	31:30	Row 3, Col 3 LUT entry
	29:28	Row 3, Col 2 LUT entry
	...	...
	3:0	Row 0, Col 1 LUT entry
	1:0	Row 0, Col 0 LUT entry
gammaLut[0]	LUT entries for applying Gamma to reference image	
	31:24	Element 3 of Gamma LUT for range [0, 31]
	23:16	Element 2 of Gamma LUT for range [0, 31]
	15:8	Element 1 of Gamma LUT for range [0, 31]
	7:0	Element 0 of Gamma LUT for range [0, 31]
gammaLut[1]	LUT entries for applying Gamma to reference image	
	31:24	Element 7 of Gamma LUT for range [0, 31]
	23:16	Element 6 of Gamma LUT for range [0, 31]
	15:8	Element 5 of Gamma LUT for range [0, 31]
	7:0	Element 4 of Gamma LUT for range [0, 31]
gammaLut[2]	LUT entries for applying Gamma to reference image	
	31:24	Element 2 of Gamma LUT for range [32,255]
	23:16	Element 1 of Gamma LUT for range [32,255]
	15:8	Element 0 of Gamma LUT for range [32,255] (this entry is ignored!)
	7:0	Element 8 of Gamma LUT for range [0, 31]
gammaLut[3]	LUT entries for applying Gamma to reference image	
	31:24	Element 6 of Gamma LUT for range [32,255]
	23:16	Element 5 of Gamma LUT for range [32,255]
	15:8	Element 4 of Gamma LUT for range [32,255]
	7:0	Element 3 of Gamma LUT for range [32,255]
gammaLut[4]	LUT entries for applying Gamma to reference image	
	15:8	Element 8 of Gamma LUT for range [32,255]
	7:0	Element 7 of Gamma LUT for range [32,255]
distCfg	31:0	Address of (Cosine 4th law) look-up table
distOffsets		distance-based (Cosine 4th law) look-up table X and Y tile offsets
	31:16	Y co-ordinate offset (unsigned)
	15:0	X co-ordinate offset (unsigned)
fullFrmDim	31:16	Full image width
	15:0	Full image height

8.10 Sharpen filter

Input	FP16/U8F
Operation	Enhanced unsharp mask. Programmable (separable, symmetric) blur filter kernel. Sharpening functionality can be disabled to use filter kernel on its own.
Filter kernel	3x3, 5x5 or 7x7
Local line buffer	Yes, maximum supported image width is 4624
Output	FP16/U8F sharpened image, filtered image or delta (mask)
Instances	1

The sharpening filter implements an enhanced unsharp mask. It addresses short-comings of the traditional unsharp mask, including:

- Undershoot and overshoot: over sharpening around strong edges, resulting in over dark and saturated pixels
- Halos around strong edges
- Amplification of noise, especially in smooth areas

It consists of a configurable blur kernel implemented as a 7x1 vertical FIR filter cascaded into a 1x7 horizontal FIR filter suitable for the implementation of any separable, symmetric function (typically a Gaussian filter is used). The size of the filter kernel is programmable from 3x3 up to 7x7. If a kernel size smaller than the maximum is used the corresponding filter coefficients should be programmed to zero. There are four programmable filter (symmetric) coefficients shared by each filter direction (vertical/horizontal). The filter coefficients are numbered 0 to 3, with 3 corresponding with the center pixel of the kernel and 0 corresponding to the (two) outermost pixels (see the UsmParam::coef01 and UsmParam::coef23 variables). The filter is implemented using FP16 arithmetic. For blur operation the sum of (all 7 of) the filter coefficients should be 1.0 (or as close as possible).

Figure 28 shows the block diagram of the sharpening filter. The basic unsharp mask operation is as follows: blur filter output is subtracted from the original input pixels to create a mask. The mask is then added back to the original image.

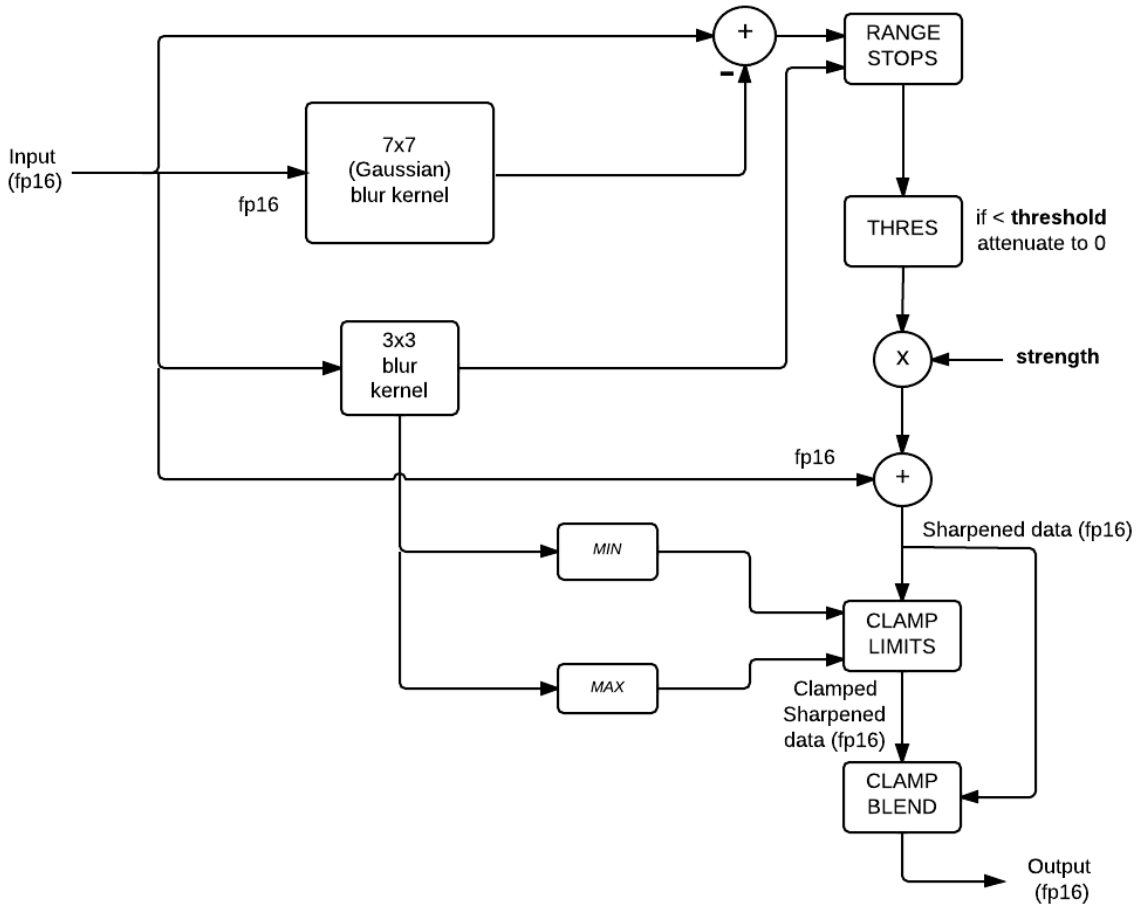


Figure 32: Sharpening filter block diagram

8.10.1 Features

8.10.1.1 Range stops

Range stops are used to apply sharpening selectively in the tonal range. For example, we normally don't want to sharpen dark areas of the image, where a lot of noise tends to be present. The most sharpening is usually desired in the mid-tones. Additionally, it is important to avoid a sudden transition between the sharpened and the non-sharpened brightness range. The range is specified using four stops (see the UsmParam::rgnStop01 and UsmParam::rgnStop23 variables). For example, let's say we wanted no sharpening in the brightness range [0, .05], full sharpening in the range [.15, .75], and less sharpening towards the saturation point. We would specify our four stops as { .05, .15, .75, 1.0 }.

The range stops are implemented using a slightly smoothed version of the input pixel, generated with a 3x3 blur kernel with the following coefficients:

```
[ [0, 1, 0],
  [1, 4, 1],
  [0, 1, 0] ] / 8.
```

8.10.1.2 Threshold, strength and delta output

After the application of the range stops the mask is set to zero if its absolute value is less than a programmable THRESHOLD (see the UsmParam::cfg variable). After thresholding the mask value is multiplied by one of two programmable strengths to give an overall DELTA which is typically added back to the original input pixel. STREN_POS is used if the input at this point is positive while STREN_NEG is used if it is negative (see the UsmParam::strength variable).

8.10.1.3 Overshoot and undershoot limits

The amount of sharpening applied may be limited by using two programmable FP16 values: OVERSHOOT and UNDERSHOOT (see the UsmParam::limit variable). If we were to specify the limit of the sharpening applied as a percentage difference from the original pixels then OVERSHOOT and UNDERSHOOT are derived as follows:

- OVERSHOOT = 1 + limit/100
- UNDERSHOOT = 1.0/OVERSHOOT

Above, UNDERSHOOT is the reciprocal of OVERSHOOT, however the two values may be programmed entirely independently: they are represented as FP16 values, overshoot should be in the range [1.0, 2.0] and undershoot should be in the range [0, 1.0]. The implementation calculates the minimum and maximum pixel values in the local 3x3 neighborhood, from the smoothed version of the image (as described above). The resulting range, [*min*, *max*], is then expanded by the specified percentage amount [UNDERSHOOT, OVERSHOOT], and output pixel is then clamped to be within that range: i.e. if the output pixel from the previous stages of the filter is greater than the *max* * OVERSHOOT, then it is set to *max* * OVERSHOOT; if the output is less than the *min* * UNDERSHOOT, then it is set to *min* *UNDERSHOOT.

8.10.1.4 Clipping Alpha

This is a function applied to the data to blend between the unclamped sharpened data and the clamped sharpened data as shown below. Alpha factor is an FP16 value which must be in the range [0, 1.0], programmed via the UsmParam::clip variable.

```
blend = (clamped sharpened data * alpha) +
            (sharpened data* (1-clipping_alpha))
```

8.10.1.5 Bypass Modes

The sharpening functionality my be disabled and the filter may be used as a stand-alone blur kernel by setting bit 4 of the UsmParam::cfg variable. In this mode the output of the programmable symmetric blur kernel is written to the output buffer, rather than the sharpened image. Alternatively, the unsharp mask (following the application of range stops, thresholding and strength multiplier) may be selected for output instead by setting bit 5 of the UsmParam::cfg variable.

8.10.2 Configuration

This filter is configured via the ***UsmParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	31:16	Threshold. Unsharp mask is set to zero if it's absolute value is less than this threshold (FP 16)
	5	Output deltas only. Set to 1 to output the internally generated deltas

Name	Bits	Description
		otherwise output input + delta
	4	Sharpening filter mode. Set to 1 to bypass sharpening and use as generic, symmetric, separable 7x7 filter kernel (e.g. for blur)
	3	Output clamp. Set to 1 to clamp the filter FP16 output into the range [0, 1.0]
	2:0	Kernel size. Configures width and height of pixel kernel (3 <= KERNEL_SIZE <=7).
strength	31:16	Negative sharpening strength (FP16)
	15:0	Positive sharpening strength (FP16)
clip	15:0	Sharpening Clipping Alpha, ALPHA (FP16)
limit	Sharpening filter limits	
	31:16	Overshoot – FP16 in the range [1.0, 2.0]
	15:0	Undershoot – FP16 in the range [0.0, 1.0]
rgnStop01	Sharpening filter range stops	
	31:16	Range stop 1 (FP16)
	15:0	Range stop 0 (FP16)
rgnStop23	Sharpening filter range stops	
	31:16	Range stop 3 (FP16)
	15:0	Range stop 2 (FP16)
coef01	Sharpening filter blur kernel coefficients	
		Gaussian filter coefficient 1 (FP16)
		Gaussian filter coefficient 0 (FP16)
coef23	Sharpening filter blur kernel coefficients	
		Gaussian filter coefficient 3 (FP16)
		Gaussian filter coefficient 2 (FP16)

8.11 Chroma Generation Filter

Input	3 planes (in parallel) of U8/U16 RGB data
Operation	- Spatial sub-sampling to half the resolution in each dimension - Reduction of Purple Flare artifacts - Desaturation of dark areas - Generation of Chroma (colour ratio) data
Filter kernel	3x3
Local line buffer	Yes, maximum supported (input) image width is 4624

Output	3 planes of sub-sampled U8 Chroma data
Instances	1

The Chroma Gen filter performs a number of functions:

- Reduces the image size by one half in each dimension.
- Reduces Purple Flare.
- Desaturates dark areas.
- Generates Chrominance data in U8 format.

The input to the filter is RGB. The bit depth of the RGB is configurable via the GenChrParam::cfg variable. The filter data-path itself is 12 bit and the input data is mapped into this range.

NOTE: When the filter is part of the oPipe however and is directly connected to streaming RGB output from Bayer demosaicing a bit depth of 12 bits must be used. The output bit depth of the Bayer demosaicing filter must be configured to match.

Downsizing is performed by simple averaging of every 4 pixels in a 2x2 block.

8.11.1 Features

8.11.1.1 Purple Flare reduction

The purple flare reduction filter modifies the blue channel only, performing a constrained sharpening operation on this channel. The strength of the correction is programmable.

8.11.1.2 Dark area saturation

Dark area desaturation removes color from dark areas, pushing them towards gray. The amount of desaturation to apply is based on the pixel intensity. The user can control the range of intensity values which are desaturated, via a programmable offset and slope. The amount of desaturation to be applied, *alpha*, is calculated as follows:

$$\alpha = (max(R, G, B) - offset) * slope$$

Alpha is then clamped to the range [0, 1]. The following graph shows *alpha* as a function of max(R,G,B):

The desaturation is performed for each channel by blending between the channel value (R, G or B) and the pixel's luminance, L. L is computed as follows:

$$L = a * R + b * G + c * B \quad (1)$$

where a, b and c are programmable coefficients. The blend is performed as follows:

$$R = R * \alpha + L * (1 - \alpha)$$
$$G = G * \alpha + L * (1 - \alpha)$$
$$B = B * \alpha + L * (1 - \alpha)$$

8.11.1.3 Chroma generation

Finally, the RGB data is converted to Chrominance data, by dividing each of R, G and B by Luminance. The Chrominance is calculated as follows:

$$Cr = \frac{R}{(L + epsilon)} * Kr$$
$$Cg = \frac{G}{(L + epsilon)} * Kg$$
$$Cb = \frac{B}{(L + epsilon)} * Kb$$

Where L is calculated as per equation (1) above, Kr, Kg and Kb are programmable coefficients, and *epsilon* is a programmable 8 bit constant. Note that the programmed value of *epsilon* is mapped to a 12 bit value before being used in the arithmetic, as follows:

```
Epsilon12 = (Epsilon8 << 4) | (Epsilon8 >> 4)
```

The final 12 bit value of epsilon should match the 12 bit value programmed for the Color Combination filter.

8.11.2 Configuration

This filter is configured via the ***GenChrParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	30:29	Local line buffer downscale rounding mode: 00 – Propagate carry bit 01 – Don't propagate carry bit 10 – Horizontal average uses fixed carry-in of 1, vertical average uses fixed carry-in of 0
	28	Bypass (enable pass-through mode)
	27:24	Input data width – 1
	23:16	Multiplier used to calculate *alpha* value for dark area desaturation
	15:8	Offset used to calculate *alpha* value for dark area desaturation
	7:0	Strength of Purple Flare reduction. Format is U(5,3).
yCoefs	Coefficients for Luma generation	
	23:16	Weight applied to Blue channel for Luma generation
	15:8	Weight applied to Green channel for Luma generation
	7:0	Weight applied to Red channel for Luma generation
chrCoefs	Coefficients for chroma generation	
	31:24	'Kb' value used during Chroma generation
	23:16	'Kg' value used during Chroma generation
	15:8	'Kr' value used during Chroma generation

Name	Bits	Description
	7:0	'epsilon' value used during Chroma generation

8.12 Median Filter

Input	U8 or U8F data, optional Luma input buffer for Chroma median mode.
Operation	Classic 2D median filter with configurable kernel size. Supports Chroma median mode where median filter (Chroma) result is alpha blended with original data based on Luma. Planar Chroma difference data is processed sequentially in this mode.
Filter kernel	3x3, 5x5 or 7x7
Local line buffer	Yes, for Chroma median mode maximum supported sub-sampled Chroma image width is 2620 (corresponds to 4624 Luma image width).
Output	Median value or any other sorted value from min to max may be selected for output, e.g. erode/dilate operations may be implemented by selecting min/max for output, respectively.
Instances	1

The median filter implementation uses a progressively updated sorting approach. The filter kernel size is configurable as 3x3, 5x5 or 7x7 via the MedParam::cfg variable. Note that the default value of zero for the kernel size bit field is invalid – the kernel size must be explicitly set before the filter is used.

The filter consumes columns of pixels and maintains an ordered list of pixels, sorted by value. The list order is updated progressively as new columns are read into the kernel and old columns are evicted. The central pixel in the list contains the median value and is selected for output by default. It is possible to output any value in the list via the output selection bit field of the MedParam::cfg variable. This means the filter may be configured to implement erode/dilate morphology filters as described below.

A programmable threshold is also provided allowing selective filtering of pixel values; only pixels greater than the threshold value are filtered. Pixel values less than or equal to the threshold are passed through unmodified. The threshold is signed and its reset value is -1 so all pixels are filtered by default. To disable filtering entirely the threshold may be set to 255.

8.12.1 Features

8.12.1.1 Chroma median use case and luma alpha blending

In the oPipe the median filter is used to process sub-sampled 3-plane chroma. In this use case the median result may be alpha blended with the original input based on the local Luma intensity such that in low light conditions, where the chroma information is less reliable, the results are pulled towards the median, but where lighting conditions are good the filter effect is weaker. The Luma alpha blend is performed as follows:

$$alpha = (\text{sub-sampled}(luma) + offset) * slope$$

$$output = orig * alpha + median * (1 - alpha)$$

Where offset and slope are programmable via the MedParam::lumaAlpha variable. Offset is a signed 8 bit value S(8,0), slope is an unsigned 8 bit value U(8,0).

Reading of and blending with the luma plane must be specifically enabled for this use case, via the MedParam::cfg variable. The luma plane may be in FP16 or U8F formats. If it is in FP16 (as indicated by the

FORMAT bit field of its MedParam:: cfg variable) then it is converted to U8F on input to the filter data-path. If the luma plane is not also sub-sampled then it may be sub-sampled by applying the following settings:

- Set the luma sub-sample bit of the MedParam::cfg variable register to 1, this enables horizontal sub-sampling by skipping pixels and sub-sampling vertically by having the filter issue two luma input buffer fill level decrements after each run.
- The luma input buffer line stride LS should be set to 2x its full resolution value.
- The luma input buffer's number of planes NP should be set 2 (i.e. 3 planes, to match the number of planes of chroma data being processed sequentially).
- The luma input buffer's plane stride PS should be set to 0, this has the effect of the forcing the read client to read the same plane repeatedly (sequentially, when NP > 0).

8.12.1.2 Support for morphological operations

The median filter may be configured to implement two useful morphological operations: erode and dilate.

To implement an erode morphology filter the value of the output pixel should be the minimum value of all the pixels in the input pixel's neighborhood, i.e. for a 7x7 kernel, set the output selection to zero.

To implement dilate morphology filter the value of the output pixel should be the maximum value of all the pixels in the input pixel's neighborhood, i.e. set the selection to the size of the filter kernel minus 1, so 48 for a 7x7 kernel.

8.12.2 Configuration

This filter is configured via the ***MedParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	30	Set this bit to 1 to decouple read and write access to the local line buffer. This is only possible if the programmed kernel size is less than the maximum supported kernel size.
	29	Set this bit to 1 to enable horizontal and vertical sub-sampling of the luma input data used for luma alpha blending
	28	Set this bit to 1 to enable luma alpha blending for the chroma median use case of the oPipe
	24:16	9-bit signed threshold. Only pixel values greater than THREHSOLD are filtered. Default value is -1 so all pixels are filtered.
	13:8	Default value of zero configures output of minimum value of kernel.
	2:0	Configures width and height of pixel kernel (3 <= KERNEL_SIZE <= 7). **Note that this must be configured correctly before use. The reset value of 0 is invalid.**
lumaAlpha	Median filter luma alpha blending control	
	15:8	Slope parameter for deriving the luma alpha blend value. Format is U(8,0).
	7:0	Offset value for deriving the luma alpha blend value. Format is S(8,0).

8.13 Chroma Denoise

Input	Up to 3 planes in parallel of sub-sampled U8 Chroma data
Operation	Chroma denoise using wide cascaded, thresholded box filters. Data is pre-filtered with a programmable 3x3 convolution then the main filtering operation is performed first vertically then horizontally on 1, 2 or 3 planes in parallel
Filter kernel	3x3 (convolution pre-filter) then up to 21x23
Local line buffer	Yes, maximum supported image width is 2320
Output	Up to 3 planes of sub-sampled U8 Chroma data
Instances	1

The Chroma Denoise filter removes color noise, including *color splotch* noise, which tends to be very low-frequency especially in low light conditions, so an extremely large filter is needed. The filter operates on planes of Chroma difference data. The filter operation is essentially that of a cascaded thresholded box filter. Filtering is performed first vertically then horizontally. The filter works by finding the average of similar pixels within the pixel's neighborhood. Similar pixels are those that within a programmable threshold.

The Filter can run in either single plane mode, or multi-plane mode. Single plane mode is activated when the number of planes per cycle is one, and multi-plane mode is activated when the number of planes per cycle is two or three. The number of planes per cycle is configured via the PLANE_MODE bit field of the ChrDnsParam::cfg variable (value programmed is number of planes – 1). Single plane mode is useful for filtering single planes of image data. Two plane mode is useful for color spaces where there are two planes of chrominance data (e.g. YCbCr color space). One advantage of this mode is that both planes can be processed in parallel, which has an obvious performance benefit, but an additional benefit is that exploiting the correlation between the two planes for the purposes of computing the weights for averaging gives better results in terms of image quality. Three plane mode works in a similar way to two-plane mode, and is useful for color spaces with three color components.

In single-plane mode, planes are processed in sequence. Weights are computed independently for each plane. In multi-plane mode either two or three planes are processed concurrently. In this mode, a single set of weights is computed, by testing whether all planes are similar enough to the center pixel. This same set of weights is then used to perform the averaging on all planes.

A pre-filter operation is performed on each plane prior to running the core Chroma Denoise filter. This is a simple 3x3 convolution. The filter must be symmetric both horizontally and vertically, so only three coefficients need to be programmed: the center coefficient, the corner coefficient, and the non-corner edge coefficient.

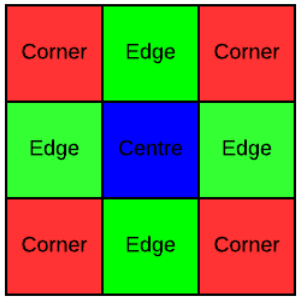


Figure 33: There are three programmable coefficients for the 3x3 convolution

After the core Chroma Denoise operation, a Grey Desaturation filter runs, which pulls pixels that are close to gray even closer. The color that represents “gray” is programmable, and therefore is not necessarily gray. For example, under warm lighting conditions, the ISP may be tuned so that photos come with a slightly warm (reddish) color cast, so that the result doesn't look too artificial. In this case, it might be desirable for the programmed “gray point” (in chroma space) to represent a slightly reddish color. Grey desaturation works by computing the difference between the current pixel and the gray point. Depending on how close they are, the color may be pulled closer to the gray point. The Grey Desaturation filter operates as follows:

$$\alpha = \text{abs}(\text{ Cr - greypoint[1]}) + \text{abs}(\text{ Cg - greypoint[2]}) + \text{abs}(\text{Cb - greypoint[3]})$$
$$\alpha = (\alpha + offset) * slope$$
$$\alpha = clamp\,01(\alpha)$$
$$C_r = greypoint[1] * (1-\alpha) + C_r * \alpha$$
$$C_g = greypoint[2] * (1-\alpha) + C_g * \alpha$$
$$C_b = greypoint[3] * (1-\alpha) + C_b * \alpha$$

The calculation of α above may be forced to 1 by setting the Grey Desat Passthrough bit in the ChrDnsParam::grayPt variable.

8.13.1 Features

8.13.1.1 Chroma difference data

Typically three chroma difference planes must be generated before processing. These are ratios of Blue to Luma, Green to Luma, and Red to Luma. An *epsilon* may be added to Luma before division to avoid division by zero. The resulting values are then scaled to the range [0, 255].

8.13.1.2 Single plane mode

In the vertical direction a 21 pixel kernel is used. The absolute differences between every pixel and the central pixel are computed. These are compared against two programmable thresholds (see the ChrDnsParam::thr[2] variables). If an absolute difference is less than the first threshold VER_T1 the pixel is given a weight of one. If an absolute difference is also less than the second threshold VER_T2 the pixel is

given a weight of two. All the pixels in the kernel are multiplied by their weights, summed and then divided by the sum of the weights. Additionally the weights can be forced to 1 for each pixel under control of software configurable bit FORCE_WEIGHTS_V (see the ChrDnsParam::cfg variable). In this case the absolute difference is not compared to the thresholds.

In the horizontal direction the same approach is used but the filtering is performed up to three times, with the output of the first pass feeding into the input of the second and so on. On the first pass a 23 pixel kernel used. The second and third passes use 17 and 13 pixel kernels, respectively. The threshold values of HOR_T1 and HOR_T2 are used in all horizontal passes instead of VER_T1 and VER_T2 respectively. The weights can also be forced to 1 for each pixel under control of software configurable bit FORCE_WEIGHTS_H.

The second and third horizontal pass will use the center pixel of the data stream output from the previous stage. Each of the horizontal passes may be individually enabled or disabled to give variable length filter.

After vertical and horizontal filtering a final step is performed: the absolute difference between the original input pixels and the filtered pixels is computed. If the absolute difference is less than a programmable threshold (the LIMIT) then the filtered pixel is output. If the absolute difference is greater than the limit, the difference between the output pixel and the input pixel is clamped at +/- LIMIT.

8.13.1.3 Multi-plane mode

In the multi-plane mode, either two or three planes are processed simultaneously. A reference image may not be used in this mode.

In the vertical direction a 21 pixel kernel is used. The absolute differences between every pixel and the central pixel of that plane are computed for each plane. These 3 differences are compared against 3 programmable threshold (VER_T1**,** VER_T2**,** VER_T3) and if all 3 are less than the threshold the pixel is given a weight of one otherwise it is given a weight of zero (if only two planes are active, VER_T3 is ignored). All the pixels in the kernel are multiplied by their weights, summed and then divided by the sum of the weights. Additionally the weights can be forced to 1 for each pixel under control of software configurable bit FORCE_WEIGHTS_V. In this case the absolute difference is not compared to the thresholds.

In the horizontal direction the same approach is used but the filtering is performed up to three times, with the output of the first pass feeding into the input of the second and so on. On the first pass a 23 pixel kernel is used. The second and third passes use 17 and 13 pixel kernels, respectively. Three Separate thresholds are set for the horizontal direction one for each plane (HOR_T1, HOR_T2, HOR_T3). The weights can also be forced to 1 for each pixel under control of software configurable bit FORCE_WEIGHTS_H.

As before each of the horizontal passes may be individually enabled or disabled to give variable length filter.

After vertical and horizontal filtering a final step is performed: the absolute difference between the original input pixels and the filtered pixels is computed. If the absolute difference is less than a programmable threshold (the LIMIT) then the filtered pixel is output. If the absolute difference is greater than the limit, the difference between the output pixel and the input pixel is clamped at +/- LIMIT.

8.13.2 Configuration

This filter is configured via the ***ChrDnsParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	31:24	Slope for Grey Desaturation – U(8,0)
	23:16	Offset for Grey Desaturation – S(8,0)
	15:14	Number of planes to process in parallel. Value programmed is number of planes – 1:

Name	Bits	Description
		0 – Single Plane per cycle (single plane mode) 1 – Two planes per cycle (multi-plane mode) 2 – Three planes per cycle (multi-plane mode) 3 – Illegal
	13	Force weights vertical
	12	Force weights horizontal
	11:4	Limit
	3	Reserved
	2:0	Horizontal filter Enable Bit 0 corresponds to first horizontal pass, bit 1 corresponds to second and so on
thr[0]	Chroma denoise 1d Weight Thresholds	
	31:24	Second Vertical threshold
	23:16	First Vertical threshold
	15:8	Second Horizontal threshold
	7:0	First Horizontal threshold
thr[0]	Chroma denoise 1d Weight Thresholds	
	23:16	Third Vertical threshold
	7:0	Third Horizontal threshold
grayPt	31	Grey desaturation passthrough. Forces α to 1 in the gray desaturation calculations, effectively disabling gray desaturation.
	23:16	Red component of color to desaturate towards
	15:8	Green component of color to desaturate towards
	7:0	Blue component of color to desaturate towards
chrCoefs	23:16	Corner filter coefficient, in U8F format
	15:8	Center-edge (non-corner-edge) filter coefficient, in U8F format
	7:0	Center filter coefficient, in U8F format

8.14 Color combination filter

Input	3 planes of sub-sampled Chroma difference data 1 plane of FP16/U8F Luma data
Operation	Upscale, and re-combine Chroma with Luma to produce RGB, color-correction matrix and offsets, 3D LUT
Filter kernel	5x5 for chroma upscale
Local line buffer	Yes, for Chroma, maximum supported sub-sampled Chroma image width is 2620 (corresponds to 4624 Luma image width)
Output	Planar RGB in FP16/U8F
Instances	1

The color combination filter recombines Chroma and Luma information back into RGB space. The Chroma planes are expected to be subsampled, so they are first upsampled by 2x in each dimension. This filter also applies the RGB2RGB matrix (which can be used to perform color correction, saturation etc.). This is a 3x3 matrix. It is followed by the addition of 3 signed offsets, which are applied independently to each of the R, B and B channels. Finally, a 16x16x16 3D LUT is applied.

8.14.1 Features

8.14.1.1 Color recombination

The “epsilon” parameter specified should be the same value that was passed when generating the Chroma data in the Chroma Gen filters. Note however that for the Chroma Gen filter, the value of Epsilon in the GenChrParam::chrCoefs variable is in range [0, 0xff], whereas for the Color Combination filter, the value of Epsilon in the ColCombParam::krgb[1] variable is in the range [0, 0xfff]. Therefore, the value programmed in the Color Combination filter should be larger by a factor of 16.

The three constants, k_r, k_g and k_b should be programmed as follows:

$$\mathrm{k\_r} = 256/K_r$$
$$\mathrm{k\_g} = 256/K_g$$
$$\mathrm{k\_b} = 256/K_b$$

where Kr, Kg and Kb are constants that would typically be equal to the constants of the same name that are programmed in the Chroma Gen filter's registers. So for example, if Kr was 85, then k_r would be 3.0118. Since the CC_K_R register field is in U(4,8) format, 3.0118*256 = 771 = 0x303 would be the value programmed.

8.14.1.2 3D LUT

The 3D LUT maps RGB values to RGB values. A user-supplied table is sampled by trilinear interpolation. The table is a 16x16x16 3D array, which each array element containing an RGB triplet of U12F values. The incoming RGB values act as co-ordinates, which select a point in 3D space within the table. The 3D LUT can be used to modify some colors in the RGB color space, while leaving others alone. This can be used to support features such as skin tone enhancement, and preferred colors/memory colors. For example, grass

can be made to look greener, and the sky can be made to look a more pleasing shade of blue.

The 3D table is a sequence of 2D images. Each 2D image is Row-major, with the incoming Red channel selecting the column (X axis), and the incoming Green channel selecting the row (Y axis). Which of the 2D images are selected to interpolate from is determined by the Blue channel (Z axis). Figure 30 illustrates the concept of the arrays and their indexing.

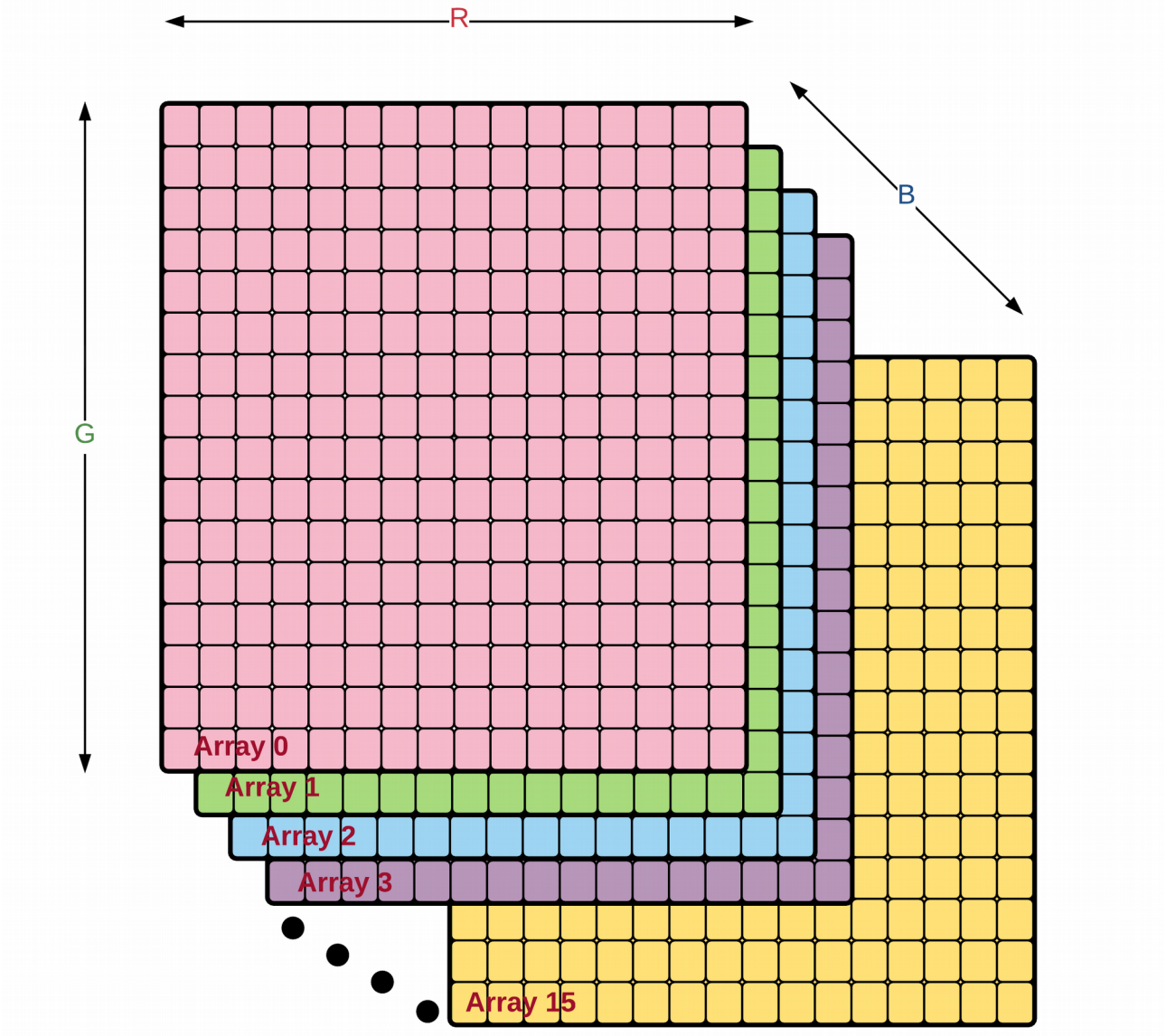


Figure 34: The Color Combination 3DLUT arrays.

The elements in each array can be uniquely indexed by using the addressing scheme shown in Figure 31. The ordering of each component (Red, Green and Blue) in each 64 bit word is also shown in Figure 31.

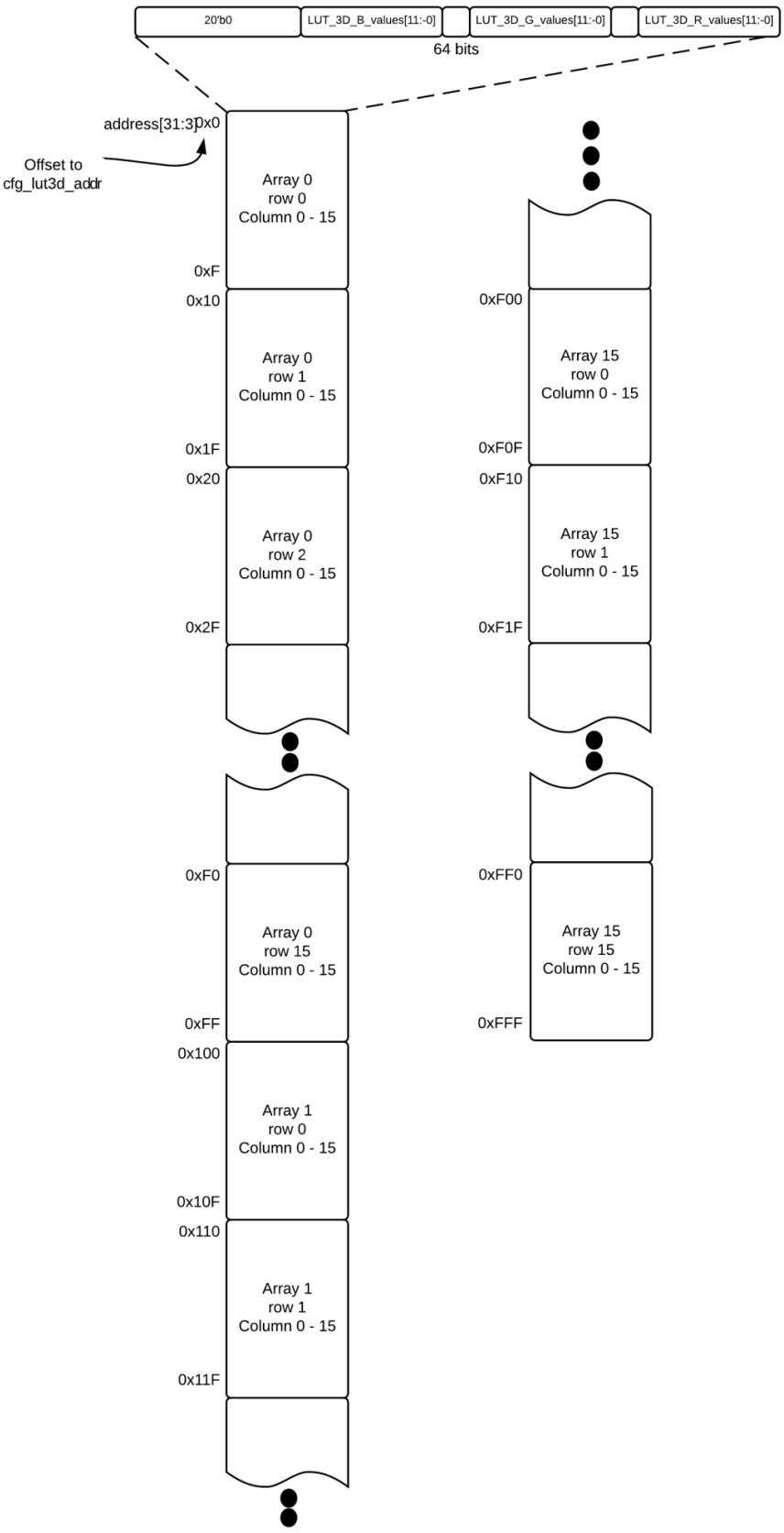


Figure 35: The memory addressing of the Color Combination 3D LUT

8.14.2 Configuration

This filter is configured via the ***ColCombParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	25:24	Plane multiple for processing multiple planes of data. Default value is for 3 output planes (RGB). Value programmed is value minus 1.
	5	Enable U12F output (rather than FP16)
	4	Clear then write to '1' to schedule a re-load of the 3D LUT memories before the next frame starts
	3	Bypass (disable) 3D LUT
	2:1	Chroma sub sampling 0 – 4:2:0 Chroma sub-sampled horizontally and vertically Other values are reserved for future use
	0	Force luma values to 1.0
krgb[0]	Color Combination K coefficients	
	27:16	Coefficient for green plane (plane 1)
	11:0	Coefficient for red plane (plane 0)
krgb[1]	Color Combination K coefficients	
	27:16	Epsilon value for all planes
	11:0	Coefficient for blue plane (plane 2)
ccm [0]	Color adjustment matrix [chroma, color]	
	31:16	CCM entry [1,0], in S(6,10) format.
	15:0	CCM entry [0,0], in S(6,10) format.
ccm [1]	Color adjustment matrix [chroma, color]	
	31:16	CCM entry [1,0], in S(6,10) format.
	15:0	CCM entry [0,2], in S(6,10) format.
ccm [2]	Color adjustment matrix [chroma, color]	
	31:16	CCM entry [1,2], in S(6,10) format.
	15:0	CCM entry [1,1], in S(6,10) format.
ccm [3]	Color adjustment matrix [chroma, color]	
	31:16	CCM entry [2,1], in S(6,10) format.
	15:0	CCM entry [2,0], in S(6,10) format.
ccm [4]	Color adjustment matrix [chroma, color] & color adjustment RED offset	
	28:16	S(1,12) offset added to Red channel after CCM is applied. Data is in the U12F format when the addition is performed.
	15:0	CCM entry [2,2], in S(6,10) format.
ccOffs	Color adjustment Green and Blue offsets	

Name	Bits	Description
	28:16	S(1,12) offset added to Green channel after CCM is applied. Data is in U12F format when the addition is performed.
	12:0	S(1,12) offset added to Blue channel after CCM is applied. Data is in U12F format when the addition is performed.
threeDLut	31:0	Address of 3D-LUT

8.15 LUT filter

Input	Up to 4 planes in parallel of FP16/U8/U16
Operation	FP16/U8/U16 pixel value look-up, e.g. for applying gamma correction/color tone mapping. Linear interpolation of nearest entries for FP16 look-up. Optional color space conversion of the output.
Filter kernel	Point operation
Local line buffer	No. Min image width 1. Min image height 1.
Output	Up to 4 planes in parallel of FP16/U8/U16
Instances	1

The look-up table filter is suitable for applying non-linear transformations to image data such as color-tone mapping or gamma correction. The filter is capable of processing both integer and FP16 data. The look-up table itself is defined as a read-only buffer. The base address of the buffer (which must be 64 bit aligned) is programmed in its base address configuration register (SIPP_IBUF[19]_BASE) and the buffer is loaded (automatically) into 16 Kb of RAM local to the filter prior to processing (i.e. at the start of the frame). The table may be re-programmed on a frame-by-frame basis with the filter updating its local RAM between frames so the processing performed may be adapted from frame to frame depending on, for example, lighting conditions or some other variant factor.

Look-up table layout

Each entry in the look-up table is 16 bits. When processing planar data either a single table may be used for all planes or one table per plane may be programmed.

Let's define a Look-up Folder (LUF) as an array of Look-up Tables (LUTs). A LUF may contain a LUT for each plane of data to be processed. All the LUTs of a LUF must be stored contiguously in memory starting with the first 16 bit entry of the first LUT, denoted LUT[0][0]. Up to 16 LUTs may be contained in a LUF and the maximum size of a single LUT is 8 Kb 16 bit entries (16 Kb, i.e. only a single maximally sized LUT).

Each LUT is organized into 16 regions. The number of LUT entries in each region is independently configurable with the restriction that it must be a power of two. The power of two size of each region, *n*, where the size of the corresponding region is 2^n, is configured via the SIPP_LUT_SIZES_7_0 and SIPP_LUT_SIZES_15_8 registers. The maximum number of entries in any single range is 1k and the minimum number of entries in any range is 1 so the programmed values of *n* can range from 0 to 10.

If the programmed power of two sizes for range[0], range[1], range[2] and so on are *n*[0], *n*[1], *n*[2] and so on, respectively, then:

Range[0] of LUT[0] starts at the programmed base address of the LUF and ends at the base address + $2^{n[0]+1}$ - 2 bytes and, Range[1] of LUT[0] starts at the programmed base address + $2^{n[0]+1}$ and ends at the base address + $2^{n[0]+1}$ + $2^{n[1]+1}$ – 2 bytes, a shown in Figure 32.

Figure 36: Look-up table size and construction

The LUT is indexed differently according to the format of the data being processed. For FP16 data the 4 least significant bits of the 5-bit FP16 exponent are used to select the LUT region then the bits of the significand are used to index the entries within the region. Note that the 4 LS bits of the exponent are sufficient to fully

cover the normalized range [0, 1.0]. If the number of entries in the region is less than 1k then only the required most significant bits of the significand are used to index the region's entries. The remaining bits of the significand (the LS bits not used to index the region), if any, form an Interpolation Numerator (IN) used to linearly interpolate between the indexed entry and the next entry. The FP16 format (also known as IEEE 754 half-precision binary floating-point format: binary16) is shown in detail in Figure 33.

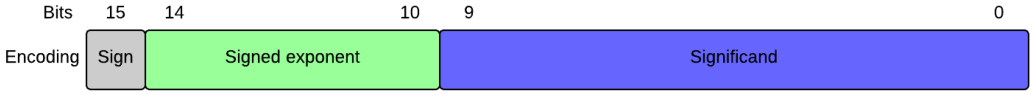


Figure 37: FP16 (IEEE 754 half-precision binary floating-point format: binary16)

For integer data the 4 most significant bits are used to select the LUT region. Thus if the integer width is 10 then bits [9:6] are used with the remaining bits used the index the region's entries. The full scheme for both FP16 and integer look-up is shown in Table 16.

Input Mode	Integer width	Range Selection	Index[9:0]
Fp16	N/A	Data[13:10]	Data[9:0]
Integer	8	Data[7:4]	{Data[3:0],6'b0}
Integer	9	Data[8:5]	{Data[4:0],5'b0}
Integer	10	Data[9:6]	{Data[5:0],4'b0}
Integer	11	Data[10:7]	{Data[6:0],3'b0}
Integer	12	Data[11:8]	{Data[7:0],2'b0}
Integer	13	Data[12:9]	{Data[8:0],1'b0}
Integer	14	Data[13:10]	Data[9:0]
Integer	15	Data[14:11]	Data[10:1]
integer	16	Data[15:12]	Data[11:2]

Table 16: LUT range selection bits

The index into each range depends on both the range size and the remaining unused bits (Index[9:0]) of the incoming data. The number of bits of FP16 significand used to index each possible range size and the number of bits left over, used for linear interpolation between entries, is shown in Table 17. It can be seen that for a maximally sized region (1 Kb entries) can be fully indexed by the 10 bit FP16 significand and that no linear interpolation between entries is performed in this case.

Pwr of 2 Region Size n	Region Size	Index bits	Interpolation Numerator
0	1	0	Index[9:0]
1	2	Index[9]	Index[8:0] << 1

Pwr of 2 Region Size n	Region Size	Index bits	Interpolation Numerator
2	4	Index[9:8]	Index[7:0] << 2
3	8	Index[9:7]	Index[6:0] << 3
4	16	Index[9:6]	Index[5:0] << 4
5	32	Index[9:5]	Index[4:0] << 5
6	64	Index[9:4]	Index[3:0] << 6
7	128	Index[9:3]	Index[2:0] << 7
8	256	Index[9:2]	Index[1:0] << 8
9	512	Index[9:1]	Index[0] << 9
10	1024	Index[9:0]	0

Table 17: Range index and interpolation fraction

8.15.1 Features

8.15.1.1 Processing planar buffers

The LUT filter provides two modes for processing planar data. The default mode is to process each plane of the input buffer in a line sequential manner (as generally supported by all filters). In this mode a separate LUT may be specified for each plane (up to a maximum of 16 planes). This allows a different look-up function to be applied to each plane. As always the entire LUF must fit within the 16 Kb local RAM. For the maximum number of planes this gives a provision of 512 entries per LUT. An example mapping of LUTs to planes is shown in Figure 34. It is also possible to use fewer LUTs than planes. In this configuration the LUTs are applied to planes on a rotating basis. For example, a 9 plane buffer could be processed on a line sequential basis using a set of 3 LUTs where LUT[0] was applied to planes 0, 3 and 6, LUT[1] was applied to planes 1, 4 and 7 and LUT [2] was applied to planes 2, 5 and 8. The number of LUTs is specified by the NUM_LUTS bit field of the LutParam::cfg variable.

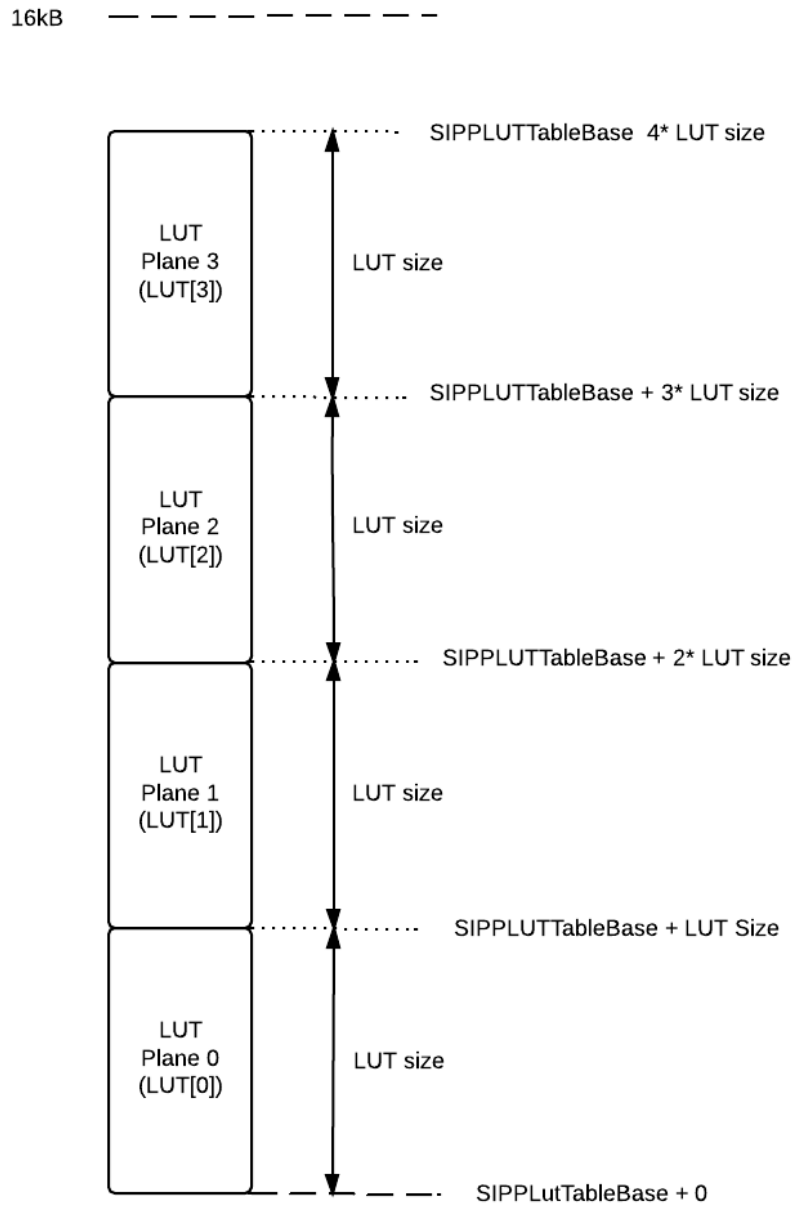


Figure 38: Multiple LUTs for sequential planar planar processing

For a higher throughput the LUT may also be configured to process up to four planes of data in parallel. This mode of processing is known as Channel Mode and is enabled by setting the LUT_CHANNEL_MODE bit of the LutParam::cfg variable. The number of planes (or channels) to process in parallel must also be specified via the NUM_CHANNELS bit field of the same register. In Channel Mode a different LUT is always applied to each channel. If it is desired to apply the same function to each plane (or some of the planes) then the same LUT should be repeated within the LUF. The NUM_LUTS bit field of the LutParam::cfg variable should be set to 0 in Channel Mode.

8.15.1.2 LUT filter configuration and LUT update

The operational mode of the LUT filter are controlled by the LutParam::cfg variable which allows the following to be configured:

- Integer or FP16 look-up (with interpolation between entries if FP16).

- Integer width (if integer look-up is configured).
- Enable and specify number of per plane LUTs for sequential multi-planar processing.

Bit 14 controls the LUT update procedure. When the LUT base address is programmed initially this bit should be set 1. If the LUT base address is changed or if the content of the LUT at the (already) programmed base address is modified and the filter will update its local RAM between frames if this bits is cleared then set to 1 again, i.e. to update the LUT filter's local RAM with new values:

- Update the LUT in system memory (CMX or DRAM).
- Update the programmed LUT base address (if necessary).
- Enable the LUT filter.
- Write '0' to bit 14 of the LutParam::cfg variable.
- Write '1' to bit 14 of the LutParam::cfg variable.
- The filter will update its local RAM at the end of frame it is currently processing.

8.15.1.3 Color space conversion

When configured in 3-plane mode, colour space conversion on the output of the LUT is supported. One output value from each plane is multiplied by a programmable 3x3 conversion matrix. A programmed offset is then added to each element of the result and the results are output to the respective planes. For example, conversion from RGB to YCbCr4:2:2 would operate as follows:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} M11 & M12 & M13 \\ M21 & M22 & M23 \\ M31 & M32 & M33 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} OF1 \\ OF2 \\ OF3 \end{bmatrix}$$

The 3x3 matrix coefficients are specified as S(2,10) values, and the offsets are in S(1,12) format. When using color space conversion, the values programmed in the LUT must be in U12F format when operating in integer mode, and in the range [0,1.0] when operating in FP16 mode.

8.15.1.4 LUT filter supported data formats

Color Space Conv	LUT Entry Format	Range	Output Format	Output
on	FP16	[0,1.0]	1	U8F
on	FP16	[0,1.0]	2	FP16
on	U12F	[0,0xfff]	1	U8F
on	U12F	[0,0xfff]	2	U12F
off	FP16	!NaN	1	U8F
off	FP16	!NaN	2	FP16
off	U12F	[0,0xfff]	2	U12F

Color Space Conv	LUT Entry Format	Range	Output Format	Output
off	U16	[0,0xffff]	2	U16
off	U8F	[0,0xff]	1	U8F

Table 18: LUT Filter Supported Data Types

8.15.2 Configuration

This filter is configured via the ***LutParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	16	Enable conversion to YUV444. Only valid in channel mode, and if the number of planes is 3.
	15	APB access Enable. For debug only
	14	Clear the set to '1' to schedule LUT update by DMA after the end of the current frame (or immediately if filter has not been run). Filter must be enabled for LUT load to take place.
	13:12	Number of channels (planes) to be processed in parallel in Channel Mode. Programmed value should be number minus 1 (maximum of 4 channels supported).
	11:8	Determines which LUT is applied to each plane when processing multiple planes sequentially. The LUT used for a given plane, P is P % NUM_LUTS. **NOTE:** If all planes are to use the same LUT, this bit field should be set to 0. **NOTE:** In Channel Mode this bit field must be set to 0.
	7:3	Integer width if in integer mode. Supports 8 to 16 bit
	1	Enable Channel Mode – if enabled up to 4 channels (planes) of data may be processed in parallel. The number of planes to be processed in parallel is specified by NUM_CHANNELS
	0	Set to 1 if FP16 interpolation required, set to 0 for integer look-up
sizeA	31:28	LUT region 7 size index n
	27:24	LUT region 6 size index n
	23:20	LUT region 5 size index n
	19:16	LUT region 4 size index n
	15:12	LUT region 3 size index n
	11:8	LUT region 2 size index n
	7:4	LUT region 1 size index n
	3:0	LUT region 0 size index n

Name	Bits	Description
sizeB	31:28	LUT region 15 size index n
	27:24	LUT region 14 size index n
	23:20	LUT region 13 size index n
	19:16	LUT region 12 size index n
	15:12	LUT region 11 size index n
	11:8	LUT region 10 size index n
	7:4	LUT region 9 size index n
	3:0	LUT region 8 size index n
lut	31:0	Address of 3D LUT
lutFormat	1:0	Format of 3D LUT 00: Invalid, 01: 8 bit format, 02 : 16 bit format, 03 : 32 bit format
mat[0]	11:0	Color conversion matrix coefficient (1,1)
mat[1]	11:0	Color conversion matrix coefficient (1,2)
mat[2]	11:0	Color conversion matrix coefficient (1,3)
mat[3]	11:0	Color conversion matrix coefficient (2,1)
mat[4]	11:0	Color conversion matrix coefficient (2,2)
mat[5]	11:0	Color conversion matrix coefficient (2,3)
mat[6]	11:0	Color conversion matrix coefficient (3,1)
mat[7]	11:0	Color conversion matrix coefficient (3,2)
mat[8]	11:0	Color conversion matrix coefficient (3,3)
offset[0]	12:0	Color conversion offset 1 in S(1,12) format
offset[1]	12:0	Color conversion offset 2 in S(1,12) format
offset[2]	12:0	Color conversion offset 3 in S(1,12) format

8.16 Polyphase Scalar

Input	FP16/U8F
Operation	Up/downscale using separable, 16-phase FIR filter
Filter kernel	3x3, 5x5 or 7x7
Local line buffer	No
Output	FP16/U8F
Instances	3

The poly-phase scaler is suitable for high-quality implementations of Lanczos or bi-cubic scaling. The filter is capable of scaling up and down and the scaling ratio is independently configurable in a both directions (horizontal and vertical). The filter implements a 16 phase FIR filter with a configurable number of taps, 3x3, 5x5 and 7x7 are possible.

8.16.1 Features

8.16.1.1 Programming filter coefficients

Filter coefficients are programmable within the range [-0.5, 1.49219]. The coefficients are programmed as 8 bit values in the range [0, 255] where the programmed value, *val*, maps to the signed fixed-point coefficient, *coeff*, as follows:

```
val = (coeff * 128) + 64
```

Programmed values of 0x40 and 0xc0 therefore correspond to a coefficients of 0.0 and 1.0 respectively. The filter data-path is FP16, the programmed coefficients are converted to FP16 values for operation and the filter can process either normalized FP16 or U8F input buffers.

8.16.1.2 Filter instances

here are 3 instances of the poly-phase scaler. Each instance has its own buffer configuration and control registers but two of the instances share a common set of filter configuration registers so if both filters are used they will be operating with identical input image dimensions, scaling ratios, number of phases and filter coefficients sets. The intended use-case is scaling of component video such as YUV 4:4:4 to YUV 4:2:2 or YUV 4:2:0 where on instance handles the Y plane and the other two handle the U and V planes.

8.16.1.3 Operation overview

Scaling involves an increasing or decreasing of the original sampling rate and is classically performed in three steps:

1. Up-sample the input by a factor of N using zero-insertion.
2. Interpolate the up-sampled input data by low-pass filtering.
3. Down-sample the result by a factor of D.

The poly-phase scaler is capable of implementing very high order FIR filters efficiently by taking advantage

of the fact that after up-sampling many filter inputs are zero and therefore do not contribute to the filter output. The SIPP poly-phase scaler supports effective FIR filter kernels of up to 112-taps organized as 16 phases of 7-taps.

The filter is constructed by cascading two 7-tap FIR filters. Filtering is performed first vertically then horizontally. The number of phases in use corresponds directly to the up-sampling factor (N). The maximum up-sampling factor is 16 (effectively allowing a 112-tap filter).

The pixel kernel for vertical scaling is 1x7 pixels: 1 output pixel is produced by filtering a column of pixels from 7 consecutive lines in the input image. However, only lines which will contribute to the vertically down-scaled output are actually read and filtered (i.e. some vertical filter phases are skipped based on the programmed vertical down-scaling factor – this complexity is handled automatically). The pixels of the current output line from vertical scaling are then horizontally up-scaled and filtered and down-scaled. The pixel kernel for horizontal scaling is 7x1 pixels: 1 output pixel is produced by filtering 7 consecutive pixels in a line. As with vertical scaling, only pixels which will be part of the down-scaled output are filtered before being written to the output buffer (i.e. some filter phases are skipped based on the programmed horizontal down-scaling factor).

8.16.1.4 General Setup

Separate scaling ratios may be specified for horizontal and vertical scaling. The ratios are specified using numerator/denominator pair (N/D) with the numerator in 5 bits and the denominator in 6 bits. The maximum supported up-scaling ratio is 16 (matching the number of filter phases in the implementation) so the allowed range for N is [1, 16]. The allowed range for D is [1, 63]. (See the PolyFirParam::horzD and olyFirParam::vertD variables for further details.)

For example, if we wish to down-scale the input image by a factor of ¾ in a given dimension we can set N to 3 and D to 4. We might also set N/D to 6/8 or 12/16, as long as the ratio of N/D is correct.

The horizontal and vertical filter coefficients for each phase are separately programmable.

Taking 12/16 as an example scaling ratio we are effectively specifying that we will use 12 of the 16 available phases to low pass filter the up-scaled (zero inserted) input image. With 7 taps per phase we are therefore implementing a 12x7 = 84 tap FIR filter. The filter function (e.g. Lanczos) should be sampled 84 times to generate the necessary coefficients. The coefficients from 0 to 83 should be programmed as follows:

```
Coefficient 0 is phase 0, coefficient 0,
Coefficient 1 is phase 1, coefficient 0,
Coefficient 2 is phase 2, coefficient 0,
…
Coefficient 11 is phase 11, coefficient 0,
Coefficient 12 is phase 0, coefficient 1,
Coefficient 13 is phase 1, coefficient 1,
…
Coefficient 82 is phase 10, coefficient 6,
Coefficient 83 is phase 11, coefficient 6.
```

The coefficients of phases 12 to 15 will not be used.

The output image dimensions must be programmed to correspond exactly to the programmed vertical and horizontal scaling factors as follows:

```
Output width = ((Input width * HorzN) - 1)/HorzD + 1
Output height = ((Input height * VertN) - 1)/VertD + 1
```

Where HorzN and HorzD are the numerator and denominator of the horizontal scale factor and VertN and

VertD are the numerator and denominator of the vertical scale factor, respectively.

The number of filter taps used in both directions (the kernel size) is configurable to 3x3, 5x5 or 7x7. If fewer than 7 taps are used then the filter coefficients for the unused taps should be set to zero. E.g. if 5 taps are used coefficients 0 and 6 must be set to zero; if 3 taps are used coefficients 0, 1, 5 and 6 must be set to zero. Note that the default value of zero for the kernel size bit field is invalid – the kernel size must be explicitly set before the filter is used.

8.16.2 Configuration

This filter is configured via the ***PolyFirParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
clamp	0	Output clamp. Set to 1 to clamp the filter FP16 output into the range [0, 1.0]
horzD	5:0	Horizontal scale factor denominator. Maximum valid value is 63
horzN	4:0	Horizontal scale factor numerator. Maximum valid value is 16
vertD	5:0	Vertical scale factor denominator. Maximum valid value is 63
vertN	4:0	Vertical scale factor numerator. Maximum valid value is 16
horzCoefs	31:0	Address of horizontal coefficients
vertCoefs	31:0	Address of vertical coefficients

8.17 Edge Operator filter

Input	U8/U16/FP16 image data or precomputed gradients
Operation	Flexible edge-detection operator suitable for implementation of e.g. Sobel filter
Filter kernel	3x3
Local line buffer	No
Output	U8/U16/FP16 edge data: magnitude + angle suitable for Histogram of Orientated Gradients
Instances	1

The Edge Operator filter implements an extension to the functionality of the standard 3x3 Sobel filter. The filter computes the X and Y gradients (G_x and G_y) using a pair of programmable Sobel filter kernels then the magnitude, M, and angle, Theta, of the edge at every pixel location is approximated as described in the following sections.

8.17.1 Features

8.17.1.1 Gradient computation

The Sobel filter kernel used to compute horizontal and vertical gradients is programmable via the EdgeParam::xCoeff and EdgeParam::yCoeff variables. The horizontal and vertical gradient computations are performed as shown below for each pixel, using *Equation 1* and *Equation 2*, respectively, where A is the 3x3 pixel kernel:

$$G_x = \begin{pmatrix} XCoeff_a & 0 & XCoeff_b \\ XCoeff_c & 0 & XCoeff_d \\ XCoeff_e & 0 & XCoeff_f \end{pmatrix} \cdot A \qquad \textit{Equation 1}$$

$$G_y = \begin{pmatrix} YCoeff_a & XCoeff_b & YCoeff_c \\ 0 & 0 & 0 \\ XCoeff_d & XCoeff_e & XCoeff_f \end{pmatrix} \cdot A \qquad \textit{Equation 2}$$

Alternatively the Sobel filter operation may be bypassed and pairs of precomputed FP16 or U8 gradients may be processed directly from the input buffer by selecting the appropriate input mode via the EdgeParam::cfg variable.

8.17.1.2 Magnitude approximation

The magnitude M is the square root of the sum of the squares of the gradients G_x and G_y In order to calculate the square root function the following manipulations are used:

$$M = \sqrt{X^2 + Y^2} = |X|\sqrt{1 + (Y/X)^2} \qquad \textit{Equation 3}$$

$$M = \sqrt{X^2 + Y^2} = |Y|\sqrt{1 + (X/Y)^2} \qquad \textit{Equation 4}$$

Now if *Equation 3* is used when $|X| > |Y|$ and *Equation 4* is used when $|X| \leq |Y|$ then an

approximation for the function $\{f(a)=\sqrt{1+a^2}\ where\ 0<a\leq 1\}$ can be developed and used to approximate for M. The following approximation is used:

The range of a (i.e. 0 to 1) is divided into 32 equal sub ranges. $f(a)$ is calculated for the top and bottom of each of theses ranges as shown in Table 19, below.

Position	a	a^2	$1+a^2$	$\sqrt{1+a^2}$
0	0.0000	0.0000	1.0000	1.0000
1	0.0245	0.0006	1.0006	1.0003
2	0.0491	0.0024	1.0024	1.0012
3	0.0736	0.0054	1.0054	1.0027
4	0.0984	0.0097	1.0097	1.0048
5	0.1233	0.0152	1.0152	1.0076
6	0.1483	0.0220	1.0220	1.0109
7	0.1735	0.0301	1.0301	1.0149
8	0.1989	0.0396	1.0396	1.0196
9	0.2246	0.0504	1.0504	1.0249
10	0.2505	0.0627	1.0627	1.0309
11	0.2767	0.0766	1.0766	1.0376
12	0.3033	0.0920	1.0920	1.0450
13	0.3304	0.1091	1.1091	1.0532
14	0.3578	0.1280	1.1280	1.0621
15	0.3857	0.1488	1.1488	1.0718
16	0.4142	0.1716	1.1716	1.0824
17	0.4433	0.1965	1.1965	1.0938
18	0.4730	0.2237	1.2237	1.1062
19	0.5034	0.2534	1.2534	1.1195
20	0.5345	0.2857	1.2857	1.1339
21	0.5665	0.3209	1.3209	1.1493
22	0.5994	0.3593	1.3593	1.1659
23	0.6332	0.4010	1.4010	1.1836
24	0.6682	0.4465	1.4465	1.2027
25	0.7043	0.4960	1.4960	1.2231
26	0.7417	0.5500	1.5500	1.2450
27	0.7804	0.6090	1.6090	1.2685
28	0.8207	0.6735	1.6735	1.2936

Position	a	a^2	$1+a^2$	$\sqrt{1+a^2}$
29	0.8626	0.7441	1.7441	1.3206
30	0.9063	0.8215	1.8215	1.3496
31	0.9521	0.9065	1.9065	1.3807

Table 19: Edge operator magnitude calculation square root approximation LUT

The values for $\sqrt{1+a^2}$ are stored in a look-up table in FP16 format. A linear interpolation between table entries is performed for values of a which lie within the range. The final magnitude may be scaled by a programmable magnitude scale factor (see EdgeParam::cfg the register).

8.17.1.3 Angle approximation (Theta)

The Angle $absolute\,\theta$ will ultimately lie between 0° and 360° depending on the magnitude and polarity of G_x and G_y. First an estimation of the primary angle $primary\,\theta$ based on $|X|$ and $|Y|$ is made as shown in Figure 39 and *Equation 5* and *Equation 6*.

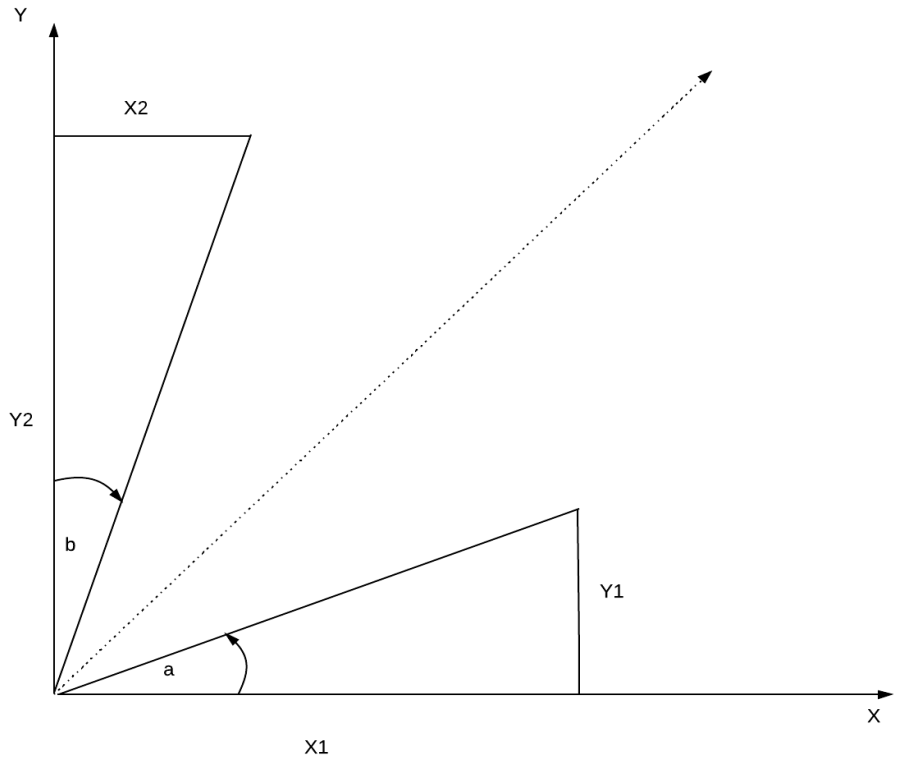


Figure 39: Edge operator primary angle

$$\theta = a, |X| > |Y| \qquad \textit{Equation 5}$$

$$\theta = b, |X| \leq |Y| \qquad \textit{Equation 6}$$

The primary angle will always lie between 0^0 and 45^0. Angles in the range 0^0 and 45^0 are represented as integers 0 to 31, giving an accuracy of 1.40625^0. The ranges are selected by means of a look-up table referenced by tan (θ), computed using *Equation 7* and *Equation 8*. The look-up table values are shown in Table 20.

$$\tan(\theta) = N/D = |Y|/|X| \; when \; |X| > |Y| \qquad \textit{Equation 7}$$

$$\tan(\theta) = N/D = |X|/|Y| \; when \; |Y| > |X| \qquad \textit{Equation 8}$$

θ >=	θ <	tan(θ) >=	tan(θ) <	Position
0.00	1.41	0.0000	0.0245	0
1.41	2.81	0.0245	0.0491	1
2.81	4.22	0.0491	0.0738	2
4.22	5.63	0.0738	0.0985	3
5.63	7.03	0.0985	0.1233	4
7.03	8.44	0.1233	0.1483	5
8.44	9.84	0.1483	0.1735	6
9.84	11.25	0.1735	0.1989	7
11.25	12.66	0.1989	0.2246	8
12.66	14.06	0.2246	0.2505	9
14.06	15.47	0.2505	0.2767	10
15.47	16.88	0.2767	0.3033	11
16.88	18.28	0.3033	0.3304	12
18.28	19.69	0.3304	0.3578	13
19.69	21.09	0.3578	0.3857	14
21.09	22.50	0.3857	0.4142	15
22.50	23.91	0.4142	0.4433	16
23.91	25.31	0.4433	0.4730	17
25.31	26.72	0.4730	0.5034	18
26.72	28.13	0.5034	0.5345	19
28.13	29.53	0.5345	0.5665	20
29.53	30.94	0.5665	0.5994	21
30.94	32.34	0.5994	0.6332	22
32.34	33.75	0.6332	0.6682	23
33.75	35.16	0.6682	0.7043	24
35.16	36.56	0.7043	0.7417	25

θ >=	θ <	$\tan(\theta)$ >=	$\tan(\theta)$ <	Position
36.56	37.97	0.7417	0.7804	26
37.97	39.38	0.7804	0.8207	27
39.38	40.78	0.8207	0.8626	28
40.78	42.19	0.8626	0.9063	29
42.19	43.59	0.9063	0.9521	30
43.59	45.00	0.9521	1.0000	31

Table 20: Edge operator arctan LUT

The absolute position $absolute\,\theta$ represents an angle between 0° and 360°. There are 256 ranges of 1.40625° between 0 and 360°. These are divided into 8 distinct regions as shown in Figure 40.

- In regions B, C, F, G $|Y|>|X|$ while in all other areas $|X|>|Y|$
- In regions C, D, E, F the polarity of X is negative while in all other areas it is positive
- In regions E, F, G, H the polarity of Y is negative while in all other areas it is positive

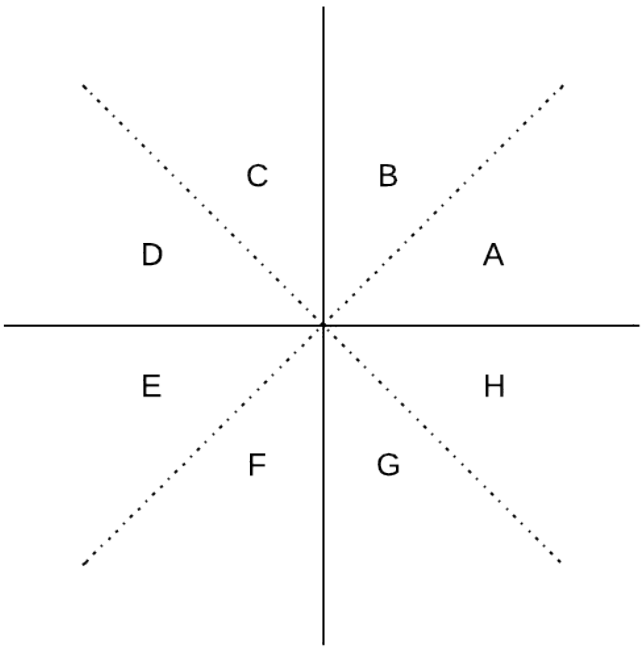


Figure 40: 8 regions used in edge operator arctan calculation

The absolute position $absolute\,\theta$ is calculated from the primary angle $primary\,\theta$ as follows:

$$absolute\,\theta = primary\,\theta, |X| \geq |Y|, X \geq 0, Y \geq 0$$

$$absolute\,\theta = 63 - primary\,\theta, |Y| > |X|, X \geq 0, Y \geq 0$$

$$absolute\,\theta = 64 + primary\,\theta, |Y| > |X|, X < 0, Y \geq 0$$

$$absolute\,\theta = 127 - primary\,\theta, |X| \geq |Y|, X < 0, Y \geq 0$$

$$absolute\,\theta = 128 + primary\,\theta, |X| \geq |Y|, X < 0, Y < 0$$
$$absolute\,\theta = 191 - primary\,\theta, |Y| > |X|, X < 0, Y < 0$$
$$absolute\,\theta = 192 + primary\,\theta, |Y| > |X|, X \geq 0, Y < 0$$
$$absolute\,\theta = 255 - primary\,\theta, |X| \geq |Y|, X \geq 0, Y < 0$$

8.17.1.4 Theta modes

There are three possible modes for the output of the angle Theta (θ). In normal mode an 8 bit value is calculated as a representation of the overall angle between 0 and 360° to an accuracy of 1.40625° (0-255) as described above.

In X-axis reflection mode all values below the X-axis are reflected in the X-axis to give the value above the X-axis. A value representing an angle between 0 and 180° to 1.40625° accuracy is output (0-127).

In X and Y axis reflection Mode all values below the X-axis are reflected in the X-axis and all values to the left of the Y-axis are reflected in the Y-axis. A value representing an angle between 0 and 90° to 1.40625° accuracy is output (0-63).

Figure 41 Illustrates where each angle segment will lie after reflection in the X and Y axis. The Theta modes are configured via the EdgeParam::cfg variable.

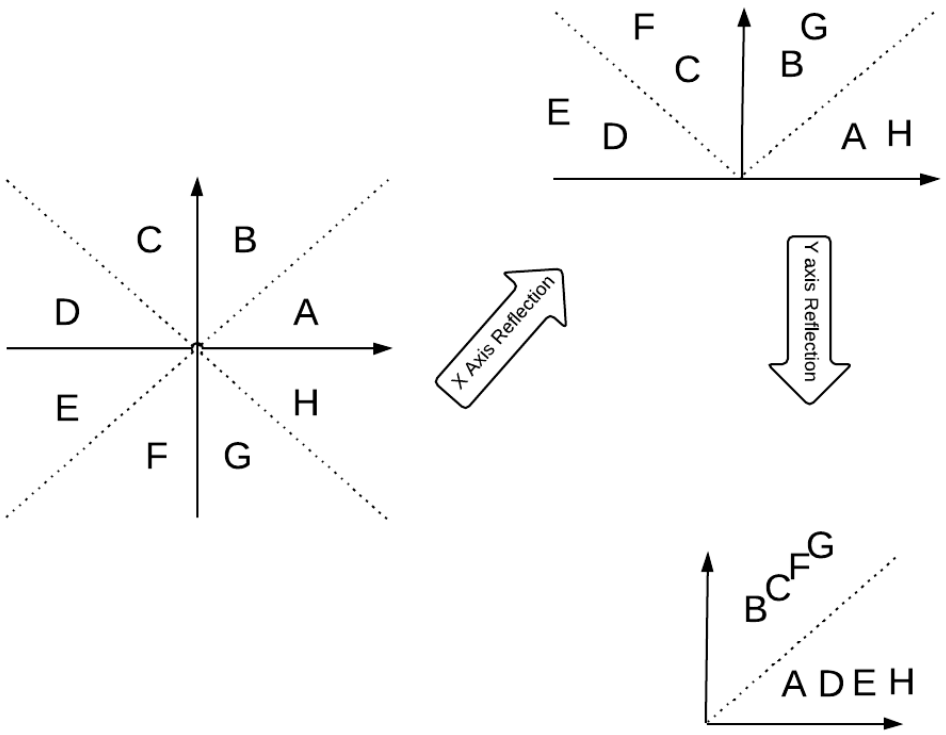


Figure 41: Edge operator Theta modes

Note for compatibility with OpenVX bit 7 of the EdgeParam::cfg register may be set to rotate theta to match the OpenVX implementation. Without setting this bit an angle of 180° in the SIPP edge operator corresponds to an OpenVX angle of 0.

8.17.2 Configuration

The Edge operator filter is configured via the ***EdgeParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	31:16	Magnitude Scale Factor. FP16 value used to scale all magnitude outputs
	7	Theta OpenVX Mode 0 – Disabled 1 – Enabled
	6:5	Theta Mode 00 – Normal mode 01 – X-axis reflection mode 10 – X and Y axis reflection mode
	4:2	Output Mode. All magnitude outputs will be scaled by the Magnitude Scale Factor 000 – 16-bit magnitude 001 – 8-bit scaled magnitude 010 – 8-bit magnitude, 8-bit angle (Theta) 011 – 8-bit angle (Theta) 100 – X, Y gradients scaled to 8 bits (2's complement) 101 – X, Y gradients output in FP16
	1:0	Input Mode 00 – Normal mode U8 pixel data 01 – Precomputed FP16 (X,Y) gradients 10 – Precomputed U8 (X,Y) gradients
xCoeff	All in signed 2's complement format	
	29:25	Edge operator X coefficient F
	24:20	Edge operator X coefficient E
	19:15	Edge operator X coefficient D
	14:10	Edge operator X coefficient C
	9:5	Edge operator X coefficient B
	4:0	Edge operator X coefficient A
yCoeff	All in signed 2's complement format	
	29:25	Edge operator Y coefficient F
	24:20	Edge operator Y coefficient E

Name	Bits	Description
	19:15	Edge operator Y coefficient D
	14:10	Edge operator Y coefficient C
	9:5	Edge operator Y coefficient B
	4:0	Edge operator Y coefficient A

8.18 Convolution filter

Input	FP16/U8F
Operation	FP16 precision convolution filter with configurable kernel size and fully programmable FP16 coefficients. Absolute or square value of results may be selected for output. Results are conditionally accumulated and counted
Filter kernel	3x3 or 5x5
Local line buffer	No
Output	FP16/U8F + FP32 accumulation and U32 accumulation count. Output may be disabled if only the accumulation/count are of interest
Instances	1

In the convolution kernel input data is convolved with a 5x5 kernel with fully programmable FP16 coefficients. Input data may be FP16 or U8F with computation is performed in FP16 precision (U8F input is scaled into normalized FP16 in the range [0, 1.0].

For programming, the coefficients of the 5x5 kernel are designated using (x, y) co-ordinates with (0, 0) at the top-left corner of the kernel and the center pixel corresponding with coefficient (2, 2), as shown in Table 21.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

Table 21: Programmable convolution kernel coefficient designation

The filter may also be configured as a 3x3 kernel (via the ConvParam::cfg variable); in this configuration the unused outer coefficients of the convolution kernel must be set to zero.

8.18.1 Features

8.18.1.1 Absolute result

The absolute value of each convolution result may be output by setting configuration option bit ABS to ‘1’.

8.18.1.2 Square result

The result of each convolution may optionally be squared by setting configuration option bit SQ to ‘1’.

8.18.1.3 Accumulate

The results may optionally be accumulated by setting the configuration bit ACCUM to '1'. If the ABS bit is set to '1' then the absolute values of the result are accumulated. If the SQ bit is set to '1' then the squares of the convolution results are accumulated. The final accumulation is performed in floating point (FP32) precision and may be read from the SIPP_CONV_ACCUM register. The accumulation runs across the frame (and for all planes if mutli-planar sequential processing is performed) but the (read-only)SIPP_CONV_ACCUM register is updated on a line-by-line basis. At the end of a frame the final accumulation value for that frame may be read until the end of processing for the first line of the next frame.

8.18.1.4 Accumulation threshold and count

A threshold THR may be set, above which results are included in the accumulation. The threshold is an FP16 value programmable via the ConvParam::cfg variable. When a result is included in the accumulation the accumulation COUNT is incremented. The final count is an unsigned 32 bit value which may be read via the (read-only) SIPP_CONV_ACCUM_CNT register. It is updated on a line-by-line basis in the same way as the accumulation itself.

8.18.1.5 Output disable

In some applications the accumulated results and count are of primary interest and the output result values themselves are not required. In these cases output may be disabled by setting configuration option bit OD to '1'. If the output is disabled then the filter's output buffer does not need to be allocated or configured.

8.18.1.6 Evan and odd coefficient sets

The filter may be configured to use a different set of coefficients on even and odd pixels by setting the EVENODD bit of the ConvParam::cfg variable to '1'. When this bit is set the filter's default coefficients are used on even pixels and its shadow coefficients are used on odd pixels.

If the EVENODD_SEL bit of the ConvParam::cfg variable is also set then on even lines the default coefficients are used on even pixels and the shadow coefficients on odd pixels but on odd lines the shadow coefficients are used on even pixels and default coefficients are on odd pixels. This configuration is suitable for processing RAW Bayer data.

8.18.2 Configuration

The convolution filter is configured via the ***ConvParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	25	Set to 1 along with EVENODD to use shadow/default coefficients on even/odd pixels of odd lines and default/shadow coefficients on even/odd pixels of even lines.
	24	Set to 1 to use different coefficient sets on even/odd pixels. Default/shadow coefficients are selected on even/odd pixels.
	23:8	Accumulation threshold (FP16 value)
	7	Output disable

Name	Bits	Description
	6	Accumulate results
	5	Output square of results
	4	Output absolute value of results
	3	Set to 1 to clamp the filter FP16 output into the range [0, 1.0]
	2:0	Configures width and height of pixel kernel. Valid values of KERNEL_SIZE are 3 and 5. **NOTE:** This must be configured correctly before use. The reset value of 0 is invalid.
kernel[0]	31:16	Coefficient (0,1)
	15:0	Coefficient (0,0)
kernel[1]	31:16	Coefficient (0,3)
	15:0	Coefficient (0,2)
kernel[2]	15:0	Coefficient (0,4)
kernel[3]	31:16	Coefficient (1,1)
	15:0	Coefficient (1,0)
kernel[4]	31:16	Coefficient (1,3)
	15:0	Coefficient (1,2)
kernel[5]	15:0	Coefficient (1,4)
kernel[6]	31:16	Coefficient (2,1)
	15:0	Coefficient (2,0)
kernel[7]	31:16	Coefficient (2,3)
	15:0	Coefficient (2,2)
kernel[8]	15:0	Coefficient (2,4)
kernel[9]	31:16	Coefficient (3,3)
	15:0	Coefficient (3,2)
kernel[10]	31:16	Coefficient (3,3)
	15:0	Coefficient (3,2)
kernel[11]	15:0	Coefficient (3,4)
kernel[12]	31:16	Coefficient (4,1)
	15:0	Coefficient (4,0)
kernel[13]	31:16	Coefficient (4,3)
	15:0	Coefficient (4,2)
kernel[14]	15:0	Coefficient (4,4), filter coefficients are FP16
NOTE : shadowkernel[15] is exactly as per kernel but for the Shadow coefficient		

8.19 Harris Corner

Input	U8
Operation	Harris corner detection with configurable kernel size and programmable *k*
Filter kernel	5x5, 7x7 or 9x9
Local line buffer	No
Output	FP32 or (scaled) FP16 scores
Instances	1

he Harris corners filter performs corner detection on U8 image data. The filter operates on a configurable kernel of 3x3, 5x5 or 7x7 pixels. The filter runs a 3x3 Sobel operator at each pixel location in the kernel. A 7x7 kernel therefore spans an area of 9x9 pixels; it is this larger size which should be used to program the HarrisParam::cfg variable. The filter computes a 2x2 gradient covariance matrix $M^{(x,y)}$ over the kernel. This matrix M is filtered by a Gaussian function (based on Pascal's triangle) then the following response is computed:

$$score(x,y) = detM^{(x,y)} - k.(traceM^{(x,y)})^2$$

The parameter k is a full-precision floating point value which is programmed via the HarrisParam::kValue variable.

The 3x3 Sobel operator used to compute the gradient covariance matrix is as follows (for dx):

```
[-1 0 1]
[-2 0 2]
[-1 0 1]
```

For dy the transpose of this same kernel is used. The score is computed to full 32 bit floating point precision.

8.19.1 Features

8.19.1.1 Output modes

Either the score or the determinant may be selected for output. Both are 32 bit floating point values. The determinant is selected for output by setting the DET_SEL bit of the HarrisParam::cfg variable to '1'. The score may be scaled by subtracting a programmable value, EXP_SUBTRAHEND, from its floating point exponent. The resulting value is known as the exponent-adjusted score. The default value of EXP_SUBTRAHEND (programmed via the HarrisParam::cfg variable) is 0. If subtracting EXP_SUBTRAHEND from the exponent would result in a negative value then the exponent of the adjusted score is clamped to 0.

The score may also be converted to an FP16 value for output. The conversion from the exponent-adjusted floating point score to FP16 is as follows:

1. Subtract (127 – 15) from the biased adjusted exponent to get the biased FP16 exponent.
2. Clamp the biased FP16 exponent in the range [0, 31].
3. Derive the FP16 significand from the 10 MSBs of the floating point significand and copy the sign bit.

Conversion of the exponent adjusted floating point score to to FP16 is enabled if the output buffer FORMAT is set to 2 (for output of the full precision floating-point score or the determinant FORMAT must be set to 4).

8.19.2 Configuration

This filter is configured via the ***HarrisParam*** structure, which has the following user-specifiable fields:

Name	Bits	Description
cfg	15:8	Exponent subtrahend. This value is subtracted from the exponent of the 32 bit floating point score before output or conversion to FP16.
	4	Select determinant for output, rather than score. FORMAT must be set to 4 when outputting the (32 bit floating point) determinant.
	3:0	Configures width and height of pixel kernel. Valid values are 5, 7 and 9. **NOTE:** This must be configured correctly before use. The reset value of 0 is invalid.
kValue	31:0	K value (FP32)

9 Filter developer's guide

Filters may be written which run on the SHAVE processors, and can be plugged into the SIPP framework. This chapter describes some concepts which must be understood before developing such a filter. It also defines the API to which SIPP filters must be written.

This should be read in conjunction with the Myriad Programmers Guide, the Myriad databook and the MvCV kernel library documentation.

9.1 Overview

Essentially, a filter's job is to produce a line of data every time it is invoked. It may reference one or more input buffers in doing so, and read one or more lines of data from each of these input buffers. It is up to the application and the SIPP framework to provide the filter with access to the required buffers. When invoked, the filter is provided with pointers to lines of input data within the input buffer.

A pointer to the location within the output buffer, where the generated line will be placed.

Usually, more than one SHAVE processor is assigned to executing a SIPP pipeline. In order to parallelize the workload, each SHAVE is assigned a portion of the scanline that it is responsible for generating.

9.2 Output buffers

Every filter has exactly one output buffer (with the exception of sink filters, which may have no output buffer). These buffers are allocated from local memory (CMX memory) by the SIPP framework. The size of the output buffer is calculated based on the type of data (e.g. 8 or 16 bits per pixel), the frame width, and the number of lines required by the filters which will consume from the output buffer. The SIPP framework also organizes the memory in the most optimal way, from a memory bandwidth efficiency point of view. The hardware provides various mechanisms to balance the memory load among the CMX slices, reducing memory access contention and avoiding stalls.

Regardless of how the memory in the line buffer is physically organized, the view of the buffer memory presented to the Shave processors is the same. The buffer is logically broken into vertical strips, with each strip assigned to one shave processor. The data may also consist of multiple planes. The diagram below shows the organization of a data buffer containing RGB data, from the SHAVEs point of view.

9.2.1 Left/right padding and replication

In Figure 42 there is padding at the left and right of each strip. This is to support operations like convolutions, which reference an area surrounding the pixel. When processing pixels near the edge of the strip, pixels may need to be referenced outside of the strip boundaries. These references access the padding areas, shown above. The SIPP framework automatically copies data into the padding areas as necessary, so that the filter does not need to perform any special handling at slice or image boundaries. The padding areas at the left of the first strip, and the right of the last strip, are populated with pixel data created by replication. Other padding areas are populated by replicating data from the neighbouring strip. The filter should not output data into the padding areas – it is generate by the SIPP framework as described. However, the generated data in the padding areas is available to be referenced by filters consuming from the buffer.

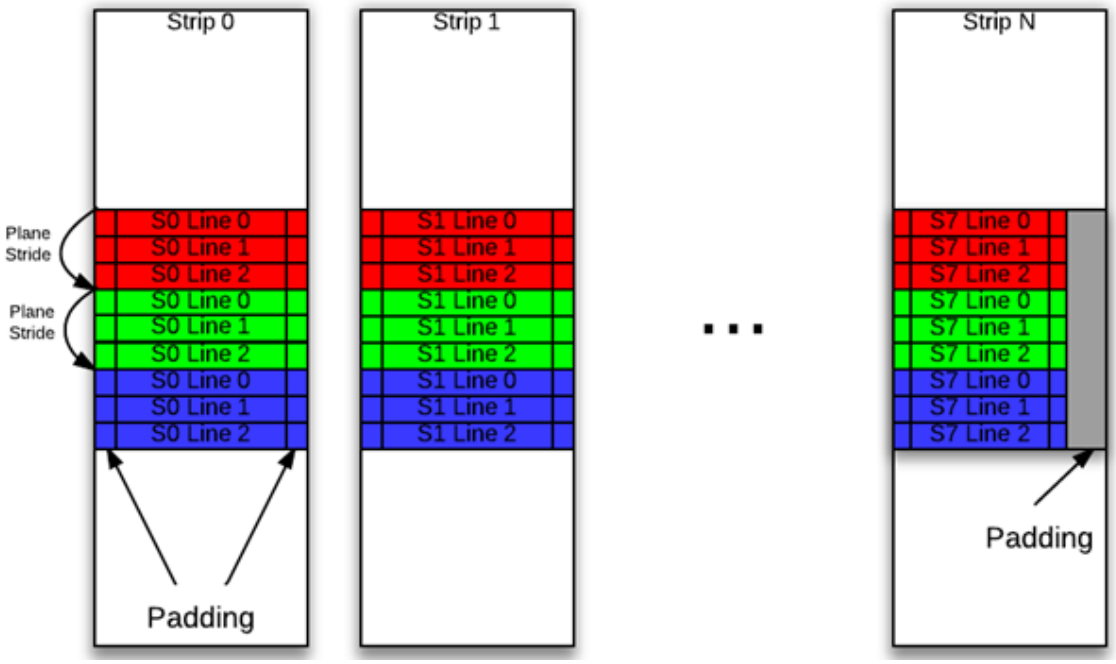


Figure 42: SIPP Buffer Padding

9.2.2 Circular buffers and line replication

SIPP buffers are implemented as circular line buffers. After a line is produced in the output buffer, the SIPP framework advances the output line pointer. When the end of the buffer is reached, the output buffer pointer is reset to the start of the buffer. Line buffer wrapping is also managed by the SIPP framework for the input buffers. The filter, when invoked, is presented with an array of line pointers for each input buffer, which point to a consecutive set of scanlines from the input image.

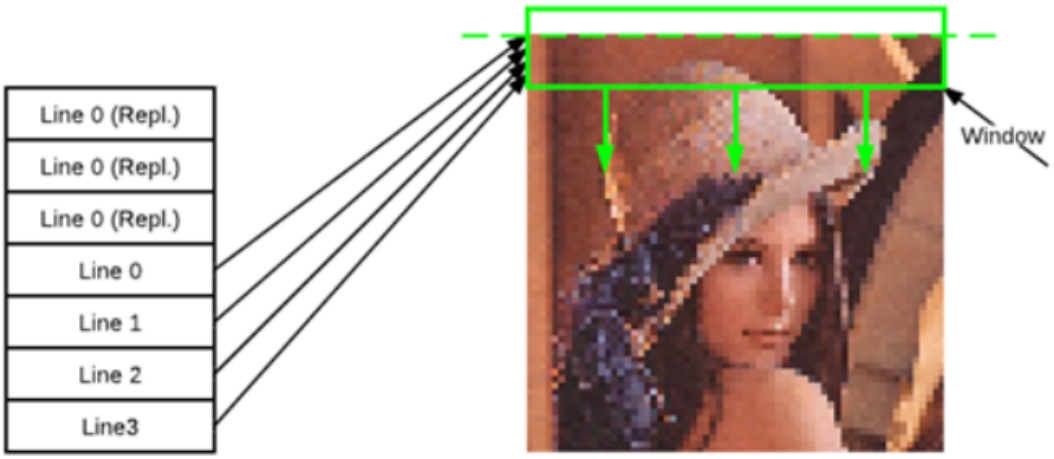


Figure 43: The SIPP framework replicates data at the top of a frame by manipulating line pointers

You can think of the line buffer as a “window” into the input frame (see diagram below). Every time the filter is invoked, the window “slides down” by one pixel before the next invocation.

In addition to the padding and replication described in the previous section, the filter does not need to do any special handling to accomplish padding by replication at the top or bottom of the frame. The SIPP framework performs virtual line replication at the top and bottom of the frame, by passing duplicate line pointers to the filters. Thus, for example, if a filter needs 7 lines of data in one of its input buffers before it can run, the first time it runs, there will be four unique lines in the buffer, and the four line pointers will point to the same memory location.

9.3 Programming language

Filters may be written using C/C++, assembly language, or a mixture of both. API calls adhere to the C calling convention in all cases. It is recommended that filters be developed in C initially. This allows them to run in a PC environment, in addition to running on Myriad hardware. After a prototype implementation has been developed in C, it is highly recommended to optimize the core of the filter code. This may be accomplished in a number of ways:

Using the vector intrinsics supported by the compiler to implement explicit vectorization.

Using inline assembly.

Implement some or all of the filter's functionality using callable routines written entirely in assembly. Assembly routines can be called from C or C++ code, if they follow the C calling convention.

The above measures can often yield an order of magnitude improvement over compiled code. Conditional compilation using the `SIPP_PC` preprocessor directive can be used so that C-only versions for PC simulations, and assembly-optimized Myriad versions, can be maintained in parallel. For example:

```
void
sampleFilter(SippFilter *fptr, int svuNo, int runNo)
{
#ifdef SIPP_PC
// C/C++ code goes here
#else
// SHAVE inline assembly code goes here
#endif
}
```

9.4 Defining a filter

Filters are self-contained, in that all header files, along with C and/or assembly files, are provided under the same folder hierarchy. Since the filters run on `SHAVE` processors, all source could should reside in a "shave" folder, as per MDK conventions.

9.4.1 Filter header file

Each filter must provide a header file, which the application must include in order to use the filter. The header file contains a function prototype for the filter's entry point, which is callable by the SIPP framework. Additionally, it defines the filter's configuration parameter structure, if any. The following is an example of a filter header file:

```
#include <sipp.h>
typedef struct
```

```
{
float strength;
}
RandNoiseParam;

void SVU_SYM(svuGenNoise)(SippFilter *fptr, int svuNo, int runNo);
```

This example filter adds random noise to an image. The filter has one configurable parameter, called "strength", which controls the level of the noise to add to the image. The prototype for all SIPP filter entry points is the same, with the exception of the function name. When this entry point is called, the filter is expected to produce one scanline's worth of data. If multiple shaves are assigned to the pipeline, then this function will be called simultaneously on all of those SHAVEs, for a given filter. Each SHAVE is only responsible for producing a portion of the pixels on the output scanline.

The filter entry point takes the following parameters:

- **ptr:** Points to the main tracking structure associated with the SIPP filter instance.
- **svuNo:** The SHAVE processors in a Myriad SOC are numbered, starting with 0. This parameter is the numerical ID of the SHAVE processing that the code is running on.
- **runNo:** during the processing of a given frame, the filter will be executed a fixed number of times, which corresponds to the filter's output frame size. Each time the filter is executed, this parameter is incremented by one. At the start of a frame, it is reset to 0.

9.4.2 The SippFilter structure

Every instance of a SIPP filter has a SippFilter structure to track it. This structure contains fields which the SIPP framework uses internally. Many of these fields however may be referenced by the filter itself. The following table documents the fields which are relevant to filter developers. The fields are described from the filter's point of view. Fields in the SippFilter structure which are not described here should not be modified or interpreted by the filter.

bpp	Bytes per pixel in the filter's output buffer. A filter may support outputting data in different formats. It could use this field to determine the expected data output format, without requiring a special configuration parameter. For example, if a filter supported U8 or U16 output data, it could reference this field to determine which type of data it was expected to produce.
nPlanes	Number of planes in the filter's output buffer.
outputW	Width of the frame to be output by this filter. This is the total number of pixels that will be produced by the filter when it runs, by all SHAVEs.
outputH	Height of the frame to be output by this filter. This is the total number of times that the filter instance will run, in the course of processing a given frame.
planeStride	When more than one plane of data is stored in the filter's output buffer, this field specifies the byte offset from the first pixel in a given plane, to the first pixel in the following plane.
params	A pointer to the filter's parameters structure. The meaning of the structure being pointed to is specific to each type of filter.
sliceWidth	When a filter runs, each SHAVE it runs on is responsible for outputting one "slice" of the total scanline. This field specifies how many pixels a single SHAVE is responsible

	for producing.
dbLinesIn	This field provides pointers to the data that the filter will operate on. For each input to a filter, one or more lines of the corresponding parent's output buffer are available to the filter to read from. This field is a three-dimensional array. The first index specifies the input. The valid range is [0, `num_inputs`], where `num_inputs` in the number of parents the filter has. The second index is used to allow parallelism, whereby the frame can do preparation work while the filter is running (the line pointers are double-buffered). The filter should always use "`runNo` & 1" as the second index. The third index specifies the line number. The valid range is [0, `num_lines`], where `num_lines` is the number of input lines the filter operates on, as specified by the application at graph creation time, via the `nLinesUsed` parameter to `sippLinkFilter()`. The lines are in top-to-bottom order. As an example, a 5x5 convolution kernel would use line pointers [0, 4] to access 5 lines of data in order to apply the convolution.
dbLineOut	This field points to the location in the filters output buffer where the filter should place the output data. Like `dbLinesIn`, a double-buffering mechanism is used to allow parallelism. The field is in fact an array of two pointers. The filter should always use "`runNo` & 1" to index the array.
gi	Global Info. Points to a "`CommInfo`" structure, which is described below.

Table 22: The SippFilter structure

The following fields of the `CommInfo` structure may be referenced by filters:

shaveFirst	First `SHAVE` assigned to the pipeline (inclusive). `SHAVE` ID's are zero-based. The `SHAVE`s assigned to the pipeline must be contiguous.
shaveLast	Last `SHAVE` assigned to the pipeline (inclusive).
curFrame	Current frame. Incremented each time the pipeline processes a frame.

Table 23: The CommInfo structure

9.4.3 SIPP Macros & Decorators

The following Macros and Decorators are defined for use in SIPP wrapper functions.

SZ	Sizeof
N_PL	Number of planes
BPP	Bits per pixel
SIPP-AUTO	Indicate that SIPP infrastructure is to allocate storage
SIPP_MBIN	Decorator to nullify argument on PC;

DDR_DATA	DDR section attribute for Myriad build; Null on PC;
ALIGNED	Alignment attribute on Myriad; Null on PC
Section	Section attribute on Myriad; NULL on PC
SVU_SYM	Mark as Myriad Shave symbol

Table 24: SIPP Argument Decorators

10 Software filters

10.1 MvCV kernels

The SIPP provides wrappers for some kernels in the MvCV kernel library. All SIPP kernels operate on lines of pixel elements.

The wrappers are located within the MDK in **common\components\sipp\filters**.

Please read the MvCV documentation on the MDK install folder for details on a particular kernel.

Table 25 illustrates some of the SW kernels available. This is **not an exhaustive list** as new kernels are added on a frequent basis.

Kernel Name	Description
absdiff	computes the absolute difference of two images
accumulateSquare	Adds the square of the source image to the accumulator.
accumulateWeighted	Calculates the weighted sum of the input image so that accumulator becomes a running average of frame sequence
arithmeticAdd	Add two arrays
arithmeticAddmask	Add with mask for two arrays
arithmeticSub	Subtract two arrays
arithmeticSubFp16ToFp16	Subtract two fp16 arrays
arithmeticSubmask	Subtract with mask for two arrays
avg	Calculate average of two arrays
bitwiseAnd	per-element bit-wise logical conjunction(AND) for two arrays
bitwiseAndMask	Per element, bit-wise logical AND for two arrays if element mask == 1
bitwiseNot	Per-element bit-wise NOT
bitwiseOr	Per-element bit-wise logical conjunction(OR) for two arrays
bitwiseOrMask	Per element, bit-wise OR for two arrays if element mask == 1
bitwiseXor	Per element, bit-wise Exclusive OR for two arrays
bitwiseXorMask	Per element, bit-wise Exclusive OR for two arrays if element mask == 1
boxFilter	Calculates average on variable kernel size, on kernel size number of input lines
boxFilterNxN	Variants of boxfilter optimized for a specific hardcoded kernel size
cannyEdgeDetection	Finds edges in the input image image and marks them in the output map using the Canny algorithm(9x9 kernel size). The smallest value between threshold1 and threshold2 is used for edge linking. The largest value is used to find initial segments of strong edges.

Kernel Name	Description
channelExtract	Extracts one of the R, G, B, plane from an interleaved RGB line.
ChromaBlock	Apply chroma desaturation and 3x3 color correction matrix.
combDecimDemosaic	–
contrast	Apply contrast on pixel element
convNxN	Convolution optimized for a specific hardcoded kernel size
convert16bppTo8bpp	Convert UInt16 pixel value to UInt8 value (clamped)
convertPFp16U16	Convert FP16 to U16
convertPU16Fp16	Convert U16 to FP16
convGeneric	Generic convolution kernel
convSeparableNxN	Optimized for symmetric matrix; Coefficients calculated externally and passed as parameter. Optimized for kernel sizes 3, 5, 7, 9 and 11 pixels.
convYuv444	Convert line to YUV444
copy	Copy elements from one line to another
cornerMinEigenVal	Calculates the minimal eigenvalue of gradient matrices for corner detection for one line
cornerMinEigenValpatched	Calculates the minimal eigenvalue for one pixel
crop	Crop image from a pixel position, Width passed as parameter
cvtColorFmtToFmt	Color conversion to various formats - NV21 to RGB - RGB to Luma - RGB to NV21 - RGB to UV - RGB to UV420 - RGB to YUV422 - YUV422 to RGB - YUV to RGB
dilateNxN	Dilates the source image using the specified structuring element which is the shape of a pixel neighborhood over which the maximum is taken. Matrix filled with 1s and 0s determines the image area to dilate on 3, 5 and 7 pixel kernel size variants of Dilate kernel.
dilateGeneric	Dilate kernel with kernel size passed as parameter. Compute the maximal pixel value overlapped by kernel and replace the image pixel in the anchor point position with that maximal value.
equalizeHist	Equalizes the histogram of a grayscale image. The brightness is normalized. As a result, the contrast is improved.
erodeNxN	3, 5 and 7 kernel size variants of Erode

Kernel Name	Description
erodeGeneric	Compute a local minimum over the area of the kernel. As the The kernel is scanned over the image, we compute the minimal pixel value overlapped by and replace the image pixel under the anchor point with that minimal value.
fast9	Feature/keypoint detection algorithm
fast9M2	Feature/keypoint detection algorithm for Myriad 2
gauss	Apply Gaussian blur/smoothing
gaussHx2_fp16	Apply downscale 2x horizontal with a Gaussian filters with kernel 5x5.
gaussVx2_fp16	Apply downscale 2x vertical with a Gaussian filters with kernel 5x5.
genChroma	Generate Chrominance from RGB input data and Luma. Variants for U8, U16, Float16 input; Used in Image Signal processing pipeline.
genDnsRef	Generate reference for hardware accelerator denoise algorithm; Used in Image Signal processing pipeline.
genLuma	Generate Luminance from RGB input; Used in Image Signal processing pipleine.
harrisResponse	Harris corner detector
histogram	Computes a histogram on a given line to be applied to all lines of an image.
homography	Homography transformation.
integralImageSqSumF32	Sum of all squared pixels before current pixel (columns to the left and rows above). Output in Float32 precision.
integralImageSqSumU32	Sum of all squared pixels before current pixel (columns to the left and rows above). Output in Unsigned 32b Integer precision.
integralImageSumF32	Sum of all pixels before current pixel (columns to the left and rows above). Output in Float32 precision.
integralImageSumU16U32	Sum of all pixels before current pixel (columns to the left and rows above). Input 16b Unsigned integer, Output in Float32 precision.
integralImageSumU32	Sum of all pixels before current pixel (columns to the left and rows above). Output in Unsigned 32b Integer precision.
laplacianNxN	Laplacian differential operator. Optimized for 3x3, 5x5, and 7x7 kernel sizes
localTM	Local Tone map operator
lowLvlCorr	Low level pixel value correction. Single plane operation.
lowLvlCorrMultiplePlanes	Low level pixel value correction. Optimized to operates on three plane (Reg, Green, Blue)

Kernel Name	Description
lumaBlur	Blur operator on luma channel.
lutNtoM	Look-up-table operator. Variants for 10 to 16, 10 to 8, 12 to 16, 12 to 8, 8 to 8.
medianFilterNxN	Median filter; Variants for 3x3, 5x5, 7x7, 9x9, 11x11, 13x13, and 15x15 resolutions.
minMaxPos	Computes the minimum and the maximum pixel value in a given input line and their position.
minMaxValue	Computes the minimum and the maximum pixel value in a given input line.
negative	Invert pixel values
positionKernel	returns the position of a given pixel value.
pyrDown	Pyramid operator using 5x5 gauss downscale operator.
randNoise	Random noise generator U8 output
randNoiseFp16	Random noise – FP16 output
sadNxN	Sum of Absolute Differences. taking the absolute difference between each pixel in the original block and the corresponding pixel in the block being used for comparison. Variants for 5x5 and 11x11 block sizes.
scale05bilinHV	Bilinear downscale filter with 0.5 factor – Horizontal and Vertical directions. Variants for U8 in/out; Fp16 in/out and Fp16 in/U8 out
scale05Lanc6HV	Apply a lanczos downscale, with factor 0.5, and 6 taps; Horizontal and vertical directions.
scale05Lanc7HV	Apply a lanczos downscale, with factor 0.5, and 7 taps; Horizontal and vertical directions.
scale2xBilinHV	Bilinear upscale – 2x, horizontal & vertical
scale2xLancH	Lanczos upscale – 2x, horizontal
scale2xLancHV	Lanczos upscaling – 2x, horizontal & vertical
scale2xLancV	Lanczos upscaling – 2x, vertical
scaleBilinArb	Bilinear scale, arbitrary X and Y scale factors
sobel	Sobel edge detection operator
ssdNxN	Sum of Squared Differences (*SSD*), the differences are squared and aggregated within a square window. Optimized for 5x5 and 11x11 window resolutions
threshold	Computes the output pixel values based on a threshold value and a threshold type: - To_Zero: values below threshold are zeroed

Kernel Name	Description
	- To_Zero_Inv: opposite of Thresh_To_Zero - To_Binary: values below threshold are zeroed and all others are saturated to pixel max value - Binary_Inv: opposite of Thresh_To_Binary - Trunc: values above threshold are given threshold value (Default type)
thresholdBinaryRange	This kernel set output to 0xFF if pixel value is in specified range, otherwise output = 0.
thresholdBinaryU8	This kernel set output to 0 if threshold value is less then input value and to 0xFF if threshold value is greater then input value
undistortBrown	Apply undistort using Brown's distortion model for known lens distortion coefficients. It supports radial (up to 2 coef.) and tangential distortions (up to 2 coef).
whiteBalanceBayerGBRG	Calculate white balance gains for Bayer GBRG input
whiteBalanceRGB	Calculate white balance gains for RGB input

Table 25: SIPP Software Kernels

Appendix A – MA2100 to MA2x5x SIPP Framework Migration

A.A SIPP MA2x5x framework

At its core the job of the SIPP MA2x5x framework remains the same as the MA2100 framework. Its task involves a graph of connected filters. Data is streamed from one filter to the next, on a scanline-by-scanline basis. Images are consumed in raster order. Scanline buffers are located in low-latency local memory (CMX). No DDR accesses should be necessary (other than accessing any pipeline input or output images located in DDR, using DMA copies to/from local memory). In addition to the performance and power benefits of avoiding DDR accesses, the design can also reduce hardware costs, allowing stacked DDR to be omitted for certain types of applications

SIPP MA2x5x framework architecture has been further motivated by a desire to free micro-processor resource from being entirely consumed by SIPP pipeline while the pipeline is operational. This has led to the creation of a nonblocking API for frame processing, expanded upon further in A.B.A.

The impact of switch to a non-blocking asynchronous API is felt to a minor degree in terms of performance. However, the reduction in performance is not significant enough to effect any but the highest throughput ISP pipelines. For these pipelines, the blocking synchronous mode API is retained (with the same limitations as on MA2100 framework) for use if desired. Further to this an alternative programming paradigm for ISP pipelines is provided within the MDK – the OPipe. The OPipe component is expected to service the highest performance ISP pipelines now.

A.B SW functionality changes

A.B.A API changes

- **Non blocking functionality**

The main feature add of the MA2x5x SIPP framework is in the addition of a non-blocking API for frame processing. This enables processing resource to be applied to other tasks when not required by the SIPP framework during the processing of the frame.

The SIPP framework operates in interrupt context during the run time for the frame. At the end of each line interrupts will be taken form constituent parts of the pipeline and a decision made on the suitability of the system to move to the next line or to complete the frame at the appropriate point.

The asynchronous behavior mandated that the framework provides a mechanism to its clients to pass back information on the progress of the pipelines. This is achieved through allowing the client to register a pipeline specific callback function. The framework will execute this callback inline when an event is raised on the associated pipeline. Since this event will most likely be raised in interrupt context, it is considered good practice for the registered callback function to do no more than set a flag indicating that the client should take care of the event in thread context at a future point.

An example of such an implementation is described in the code snippet following.

```
void appSippCallback ( SippPipeline *              pPipeline,
                       eSIPP_PIPELINE_EVENT        eEvent,
                       SIPP_PIPELINE_EVENT_DATA * ptEventData
                     )
{
    if (eEvent == eSIPP_PIPELINE_FRAME_DONE)
```

```
    {
        printf ("appSippCallback : Frame done event received : \n");
        testComplete = 1;
    }
}

int main (int argc, char *argv[])
{
    pPipe = sippCreatePipeline(0, 7, SIPP_MBIN(mbinImgSipp));
    ...

    // Register callback for async API
    sippRegisterEventCallback (pPipe,
                               appSippCallback);

    ...

    sippProcessFrameNB (pPipe);

    ...

    while ( testComplete == 0x0 )
    {

    }

    ...

    return 0;
}
```

In this situation, the client creates a pipeline and then registers a function – appSippCallback() with the framework. After calling sippProcessFrameNB the application is free to carry on with other tasks. At some future point the SIPP framework calls the registered callback function and signals that the frame has completed by passing this function the eSIPP_PIPELINE_FRAME_DONE event.

The callback function sets a flag which the client may check using some mechanism in thread context. It may then continue on to some other event on the pipeline.

- **Concurrent pipeline support**

A non blocking API allows the SIPP MA2x5x framework to present an interface enabling concurrent operations on distinct pipelines. The client is free to call sippProcessFrameNB() for all pipelines. Control over how and when each pipeline will e scheduled will ne handled by the framework.

Consider a situation in which sippProcessFrameNB is called for 4 created pipelines. A decision on which of the pipelines is to be scheduled first is made by the framework's scheduling algorithm. Once this pipeline is launched, the other pipelines will also be considered. If any is capable of running alongside the first scheduled pipeline, a further pipeline may be launched. And again the remaining pipelines will be considered in terms of their suitability for running alongside all currently scheduled pipelines.

What makes pipelines suitable for concurrent operation is naturally a lack of conflict in the resource they require. That is to say should two pipelines not share the same HW filters or desire to use the same SHAVE units, then they can rum together.

It should be noted that in the SIPP MA2x5x framework only the non blocking API will enable concurrent operation as the runtime employed y the blocking API can handle only one pipeline at a time (as per the SIPP MA2100 framework).

A.C HW filter changes between MA2100 and MA2x5x

- **Filters Removed:**
  - Bayer demosaicing post-processing median filter.
- **New Filters:**
  - Sigma Denoise:
    - Operates in Bayer domain, or on planar data.
  - Gen Chroma (RGB in, 3-plane chroma out):
    - Includes Purple Flare Reduction and Dark Area Desaturation filters.
  - Difference of Gaussians noise reduction/Local Tone Mapping (4x modes of operation):
    - DoG only: output is a thresholded difference of Gaussians (LTM filter is bypassed).
    - LTM only: output is local tone-mapped input (DoG is bypassed).
    - Denoise mode: output is input minus thresholded difference of Gaussians (LTM filter is bypassed).
    - DoG+LTM mode: output is local tone-mapped input minus thresholded difference of Gaussians.
- **Extra Scalers:**
  - Now have 3x instances of poly-phase scaler.
  - All instances are identical, however:
    - Instance 0 has its own configuration registers, but,
    - Instances 1 and 2 share a single set of configuration registers.
  - (Imagined use case is arbitrary scaling of YUV 4:4:4 to 4:2:2 or 4:2:0 where instance 0 handles Y and instances 1 and 2 handle U and V, sharing the same scaling ratio and filter coefficients.).
  - Scaler coefficients are now in a special biased 8 bit format (previously they were FP16).
- **Filters Enhanced:**
  - Harris Filter:
    - Added support for optional output of determinant.
  - Convolution Kernel:
    - Added support for even/odd coefficient sets (e.g. for Bayer processing) by borrowing shadow coefficients.
  - Edge Operator:
    - Increased precision of angle output to 1.41 degrees, i.e. angle is now represented by an unsigned 8 bit integer between 0 and 255, directly usable for indexing bins of histogram.
    - OpenVX compatibility.
  - RAW Filter:
    - Enhanced AE/AWB stats (special handling of saturated and dark pixels).
    - Added support for static defect correction (reads in defect list, handles defect rows/columns).
    - Added AF stats.
    - 256 bin luma histogram (up from 64).
    - 128 bin RGB histograms.
  - Bayer Demosaicing Filter:
    - Added luma generation (i.e. can output luma as well as RGB).
    - Added fast preview mode (downscale-demosaic).
  - Chroma Denoise:
    - Added Gaussian 3x3 pre-filter.
    - Support for 1, 2 (previously unsupported) or 3-plane operation.
    - Added optional Grey Desaturation post-filter step.
  - Luma Denoise:

    - Added reference generation block (Cosine 4th law LUT plus Gamma Adjustment).
    - Removed reference input buffer.
    - Corrected various problems missed in V1 verification.
  - Median Filter:
    - Added support for alpha blending of median filtered output with input based on luma input.
    - Added new luma input buffer.
  - LUT Filter:
    - Added support for color space conversion of output (in 3-plane mode).
    - Added requirement that LUT filter is enabled before LUT load request.
    - LUT load requests are now scheduled. Will occur after frames, or immediately if no frame start has been seen.
  - Color combination filter:
    - Removed dark area desaturation step.
    - Added 3D LUT.
  - MIPI Rx filter:
    - Added line counter and interrupt on designated line.
    - Line may be based on window line or output line.
    - Deprecated packed windows feature.
- **New Generic/Universal Features:**
  - Support for packed RAW10/12 buffers in Sigma Denoise, LSC and RAW filters.
  - Support for packed RAW10/12 input buffer in Bayer Demosaicing Filter.
  - Support for packed RAW10/12 output buffers in MIPI Rx filters.
  - Universal support for unaligned output buffers.
  - Restricted support for unaligned input buffers:
    - The following selected filter input buffers may have any alignment:
      - Median filter (primary input buffer, luma buffer must be aligned).
      - Look up table (primary input buffer, LUT buffer itself must be aligned).
      - Edge Operator.
      - Convolution Kernel.
      - Harris Filter.
      - Polyphase scalers (all instances).

A.D SIPP MA2100 to MA2x5x porting checklist

This chapter will provide a basic guide to users wishing to run existing SIPP MA2100 applications on SIPP MA2x5x framework targeted at MA2x5x.

The SIPP MA2x5x framework provides an API which is to a large degree a superset of the SIPP MA2100 framework. That is to say the SIPP MA2100 API continues to be almost fully supported in SIPP MA2x5x. This approach facilitates the most expedient route for porting existing applications to the SIPP MA2x5x framework.

As a general rule then it may be said that all SIPP MA2100 applications should build and run on SIPP MA2x5x. The exceptions to this rule are detailed in the remainder of this chapter. In brief these are broken down into a small number of API changes and the evolution of the underlying hardware on the targeted platforms.

A.E Pipeline level

One fundamental difference in the MA2x5x SIPP HW versus MA2100 is the increased efficiency of the ISP pipeline. The throughput of the system has been optimized through implementing direct connections

between HW filters. These connections are implemented via the inclusion of local line buffers (LLBs) within the filters themselves. Within these line buffers sufficient data may be held to allow one line of output from the filter to be produced at the filter's maximum operating resolution.

While the inclusion of the local line buffer aids in delivering increased performance while minimizing the power requirements of the pipelines, a consequence of their use is an increased complexity in context saving the filter status. For this reason, context switching has been dispensed with in the SIPP MA2x5x framework. This has several consequences. Chief among these are the removal of the sippProcessIters() API and the requirement that no hardware filter (bar the Polyphase Scalar) may appear twice in the same pipeline. These consequences are covered in more detail in the sections following

A.F Removal of sippProcessIters

The sippProcessIters() API allowed a sub-frame level of processing to be achieved. In turn this enabled the application to begin and carry out other tasks which were latency sensitive during the course of the processing of a full frame of data.

One of the motivations behind this (the latency of operation) has been addressed by providing a non-blocking version of the API freeing the application to begin other tasks while the SIPP framework looks after processing a full frame. Furthermore the new framework can practice real concurrency if pipelines where available. That is to say pipelines which do not have any conflict of resource in terms of SHAVEs or HW filters will be run concurrently by the new framework.

A.G No multiple use of HW filter in same pipe

The inability to simply context switch many of the HW filters of MA2x5x as stated in A.E dictates that no HW filter may be used twice in the same pipeline. Therefore for situations in which a current application has specified such a SIPP pipeline, the workaround is to decompose the existing single pipeline into more than one new pipelines, the number of new pipelines being set by the maximum number of times any single HW filter appears in the original pipeline. Each new pipeline will use the HW filter once (as well as other filters if necessary), and will leave one frame of data in DDR memory. This data then becomes the input for a second pipeline, again employing the same HW filter.

Note that multiple use of the same SW filter does not impose this constraint.

Note also that MA2x5x contains three instantiations of the Polyphase Scalar filter. Therefore these three units may each be used in a pipeline without creating the need to decompose the pipeline into multiple sub-pipes.

A.H Filter level

A.H.A Removed filters

Only the Bayer demosaicing post-processing filter has not been retained between MA2100 and MA2x5x. Clearly therefore applications which created pipelines using the Bayer demosaicing post-processing filter on MA2100 must be modified to use alternatives when targeting MA2x5x.

A.H.B Retained Filters

There are 12 filters which have persisted between MA2100 and MA2x5x. In many cases improvements in the HW design may mean that the same configuration produces slightly different results. Any bitwise checking on the output of such operations should take this into account.

In many cases the retained filters have been enhanced with additional features. Some of these features are "always on" while some are explicitly enabled. The new features are not dealt with in the context of this porting document.

The following sub-sections detail the porting considerations for each of the 12 retained HW filters in turn.

- **Harris Corner**

The Harris filter is now capable of doing its own padding. Therefore the slight cropping of the image which was necessary in MA2100 is no longer required in MA2x5x. Therefore input and output resolution should be the same.

- **Edge Detection**

There are no porting requirements for this filter. SIPP MA2100 configurations should be fully compatible with SIPP MA2x5x.

- **Convolution**

There are no porting requirements for this filter. SIPP MA2100 configurations should be fully compatible with SIPP MA2x5x.

- **Lens Shading correction**

There are no porting requirements for this filter. SIPP MA2100 configurations should be fully compatible with SIPP MA2x5x.

- **RAW**

When computing AE stats, these AE stats are now stored in a manner conforming with the following c-struct:

```
typedef struct
{
    uint32_t count    [4]; // number of pixels in alternative
accumulation.
    uint32_t accum    [4]; // accumulation of pixels within limits
    uint32_t alt_accum[4]; // accumulation of pixels outside limits

} ae_patch_stats;
```

On MA2100 these stats were just a simple uint32 array. If using the RAW filter for AE stats on MA2x5x the memory written will be larger than with MA2100. Care should be taken to ensure the memory allocation for this area reflects this.

- **Debayer**

The debayer filter is capable of producing two outputs in MA2x5x. These are RGB and Luma outputs. Each must be explicitly enabled. Therefore to port a MA2100 SIPP FW application using the debayer, the RGB output must be explicitly enabled now. This is done by setting bit 25 of the thresh member of the debayer

parameter structure as so...

```
debayer            = sippCreateFilter ( pipeLine,
                                        0x0,
                                        PIPELINE_WIDTH,
                                        PIPELINE_HEIGHT,
                                        N_PL(NUM_DEBAYER_PLANES),
                                        SZ(UInt8),
                                        SIPP_AUTO,
                                        (FnSvuRun)SIPP_DBYR_ID,
                                        0);
debayerCfg         = (DbyrParam *)debayer->params;
debayerCfg->thresh = ((0x1)<<25);
```

On MA2x5x, HW enhancements mean the debayer filter no longer crops the image. Therefore the input and output resolutions for this filter will remain the same. Existing applications do not necessarily need to be updated for this but the option is now available.

- **Median**

There are no porting requirements for this filter. SIPP MA2100 configurations should be fully compatible with SIPP MA2x5x.

- **Luma Denoise**

The luma denoise filter now has its own reference generation block. This means the reference input buffer has been removed and pipelines using this block must be suitably adjusted to remove this input.

The kernel size for this unit has also been modified to 11x11.

- **Chroma Denoise**

The chroma denoise filter is now a composite of 3 units. At the front-end there is a 3x3 Gaussian filter, followed by the main chroma denoise algorithm. Lastly there is a gray point desaturation unit.

Having the 3x3 Gaussian filter at the front means that the kernel width and height of the chroma denoise unit as used in sippLinkFilter API must be set to a kernel width of 3 and a kernel height of 3.

The 3x3 Gaussian filter may not be bypassed but of-course its effects can be eliminated through the use of the correct coefficients.

The gray point desaturation unit may be bypassed if desired but this must be explicitly added as per the code following.

```
chromadnsCfg       = sippCreateFilter ( pipeLine,
                                        0x0,
                                        PIPELINE_WIDTH,
                                        PIPELINE_HEIGHT,
                                        N_PL(NUM_CHROMADNS_PLANES),
                                        SZ(UInt8),
                                        SIPP_AUTO,
                                        (FnSvuRun)SIPP_CHROMA_ID,
                                        0);
chromadnsCfg         =  (ChrDnsParam *)chromadns->params;
chromadnsCfg->grayPt = (1 << 31);
```

➢ Sharpen

There are no porting requirements for this filter. SIPP MA2100 configurations should be fully compatible with SIPP MA2x5x.

➢ Color combination

The color combination unit has a 3D LUT included in MA2x5x. To maintain MA2100 functionality this LUT must be explicitly bypassed. This is achieved by setting bit 3 of the cfg member of the color comb parameters structure as follows:

```
colorComb          = sippCreateFilter ( pipeLine,
                                        0x0,
                                        PIPELINE_WIDTH,
                                        PIPELINE_HEIGHT,
                                        N_PL(NUM_COL_COMB_PLANES),
                                        SZ(half),
                                        SIPP_AUTO,
                                        (FnSvuRun)SIPP_CC_ID,
                                        0);
colorCombCfg       = (ColCombParam *)colorComb->params;
colorCombCfg->cfg = (1 << 3);
```

➢ Polyphase Scalar

On MA2Xx5x filter coefficients are programmed as 8 bit values in the range [0, 255] where the programmed value, *val*, maps to the signed fixed-point coefficient, *coeff*, as follows:

$$val = (coeff * 128) + 64$$

This gives a coefficient range of [-0.5, 1.49219]. Programmed values of 0x40 and 0xc0 therefore correspond to a coefficients of 0.0 and 1.0 respectively. The filter data-path is FP16, the programmed coefficients are converted to FP16 values for operation and the filter can process either normalized FP16 or U8F input buffers.

Any application using a polyphase scalar being mapped to MA2x5x needs to have the coefficient set remapped to this new range and bit width.