MA2100 SIPP User Guide

v1.0 / October 2015

Movidius Confidential

Copyright and Proprietary Information Notice

Movidius Ltd.
1730 South El Camino Real, Suite 200
San Mateo, CA 94402
http://www.movidius.com/

Revision History

Revision	Description	Author
v0.20	Draft SIPP programmers user guide + accelerator details	John Scott
v0.30	Tidy up, new template	John Scott
v0.40	Review comments; Convert to ODT format.	Team/John Scott
v0.50	Minor edits on kernel parameter	John Scott
v0.60	Removed pipeline reschedule function from API	John Scott
v0.70	Add runtime & build configuration section	John Scott
v0.80	Formatting, layout and content (diagram and table) edits. Changed title and front page. Inserted copyright page.	Daniel Grigoraș
v0.90	Added section on schedule calculation and specifiying memory area for schedule calculation	John Scott
v1.00	Added description for new API SippAllocCmxMemRegion Fixed formatting, layout, style and linguistic issues	Kyle McAdoo Daniel Grigoraș

Table of Contents

1 Introduction

1.1 About this document

This document describes the Movidius *SIPP* (Streaming Image Processing Pipeline) framework. It is targeted at users of the framework wishing to build Image Processing and Computer Vision pipelines running on Myriad 1 and Myriad 2 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 Platform Datasheet, v1.03
4. Myriad 1 Instruction Set Reference Manual ISA Revision, v1.0
5. Camera Interface Specifications, MIPI Alliance, http://www.mipi.org/specifications/camera-interface#CSI2
6. Graph Designer User Guide.

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)
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*.

AWB	Auto White 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 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, in 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 memory. As processed lines of data become available, they are transmitted directly

AWB	Auto White Balance
	from a local 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

1.4 Frame Layouts

1.4.1 Bayer

In Bayer mode, the color channels are arranged in a color filter array. The filter pattern is 50% green, 25% red and 25% blue.

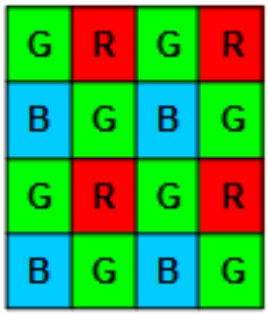

Figure 1: Example bayer color filter array

GRBG, RGGB, GBRG and BGGR bayer patterns are supported.

1.4.2 Planar

In planar mode, each color channel resides in a separate data plane.

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 Myriad 1 silicon, there are 8 `SHAVE` vector processing cores, whereas on Myriad 2, there are 12. Both SOCs have DMA controllers, for transferring data from DDR to CMX, and vice-versa. Additionally, the Myriad 2 processor has a number of hardware accelerators, for some computationally expensive ISP and computer vision tasks.

Whereas the bulk of the processing is performed by the `SHAVE` cores (and also by the hardware accelerators on Myriad 2), the SIPP framework (pipeline management and scheduling etc.) runs on a RISC processor.

The SIPP environment is also the development framework for the Myriad 2 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. 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 exactly one *output buffer*. 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 exactly one new line of data in its output buffer. 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 Graph validity

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 4: An example of a disconnected graph).
- Graph creation

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

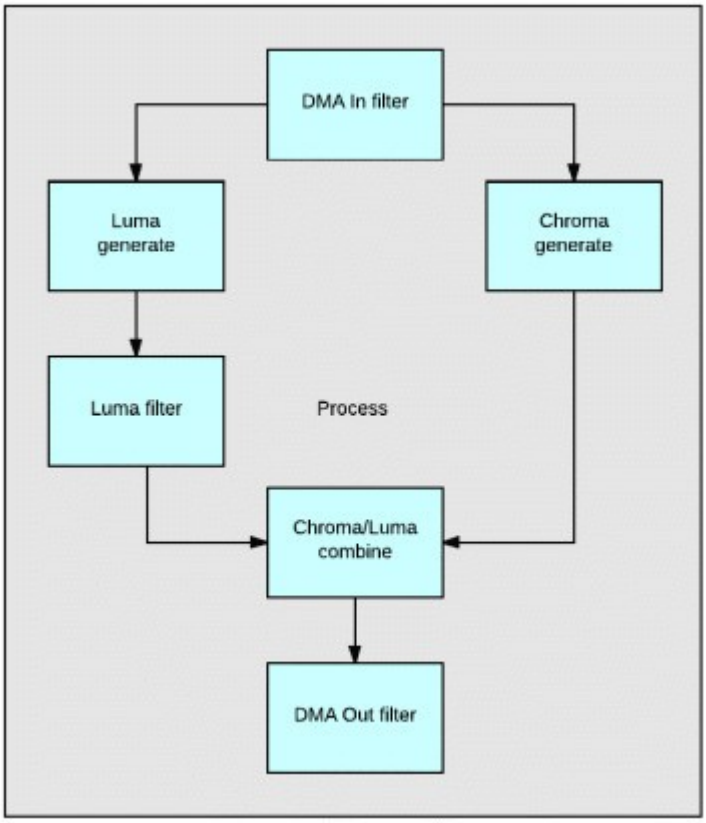

Figure 2: Simple pipeline

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.

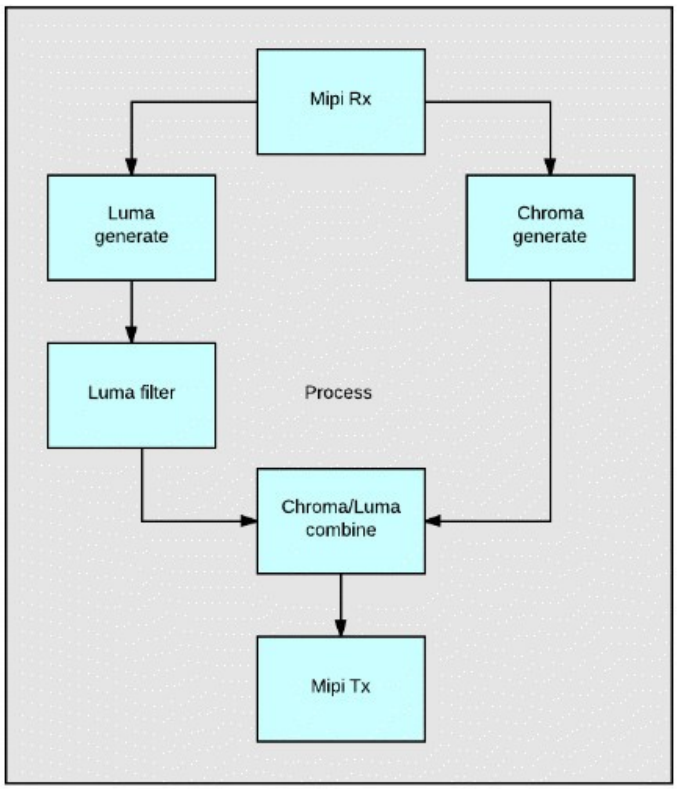

Figure 3: Simple pipeline without DDR, streaming from MIPI sensor

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.

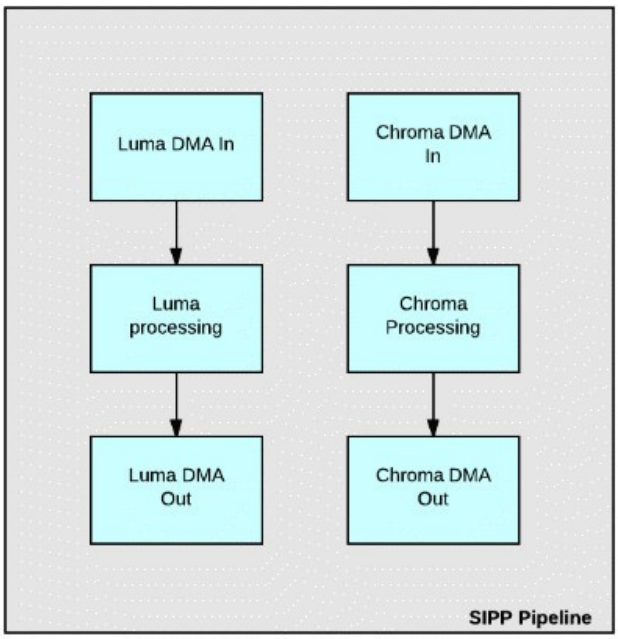

Figure 4: Example of a Superpipe - Disconnected Graph

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 produces 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.

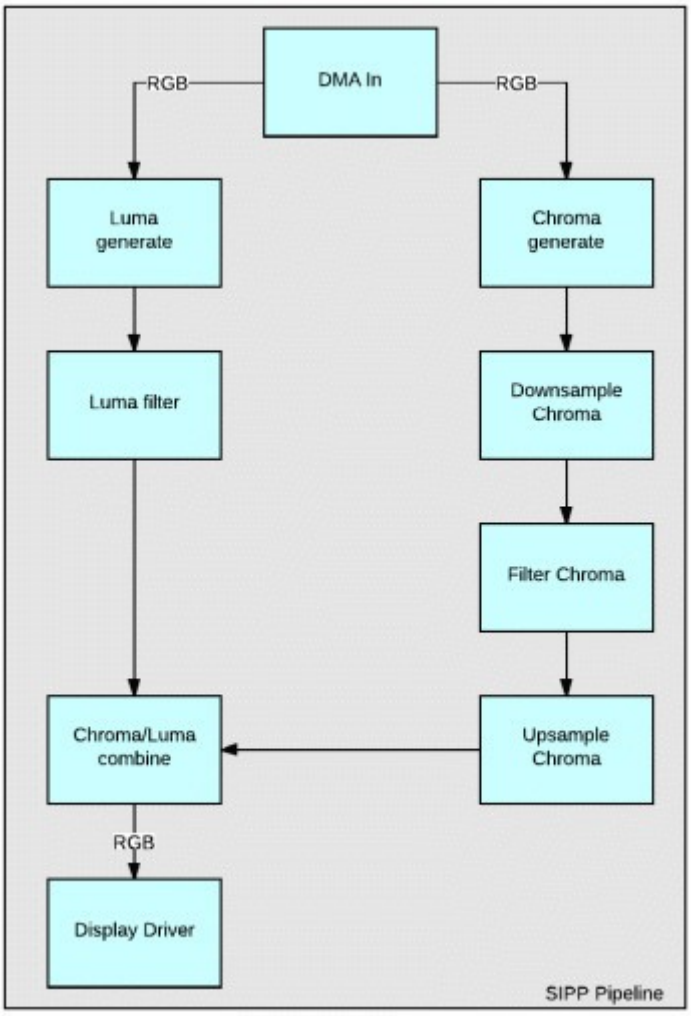

Figure 5: Filters with different output line rates

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 vs. 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 filter, whether hardware or software, can be used in a pipeline.

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 stackedDDR 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.

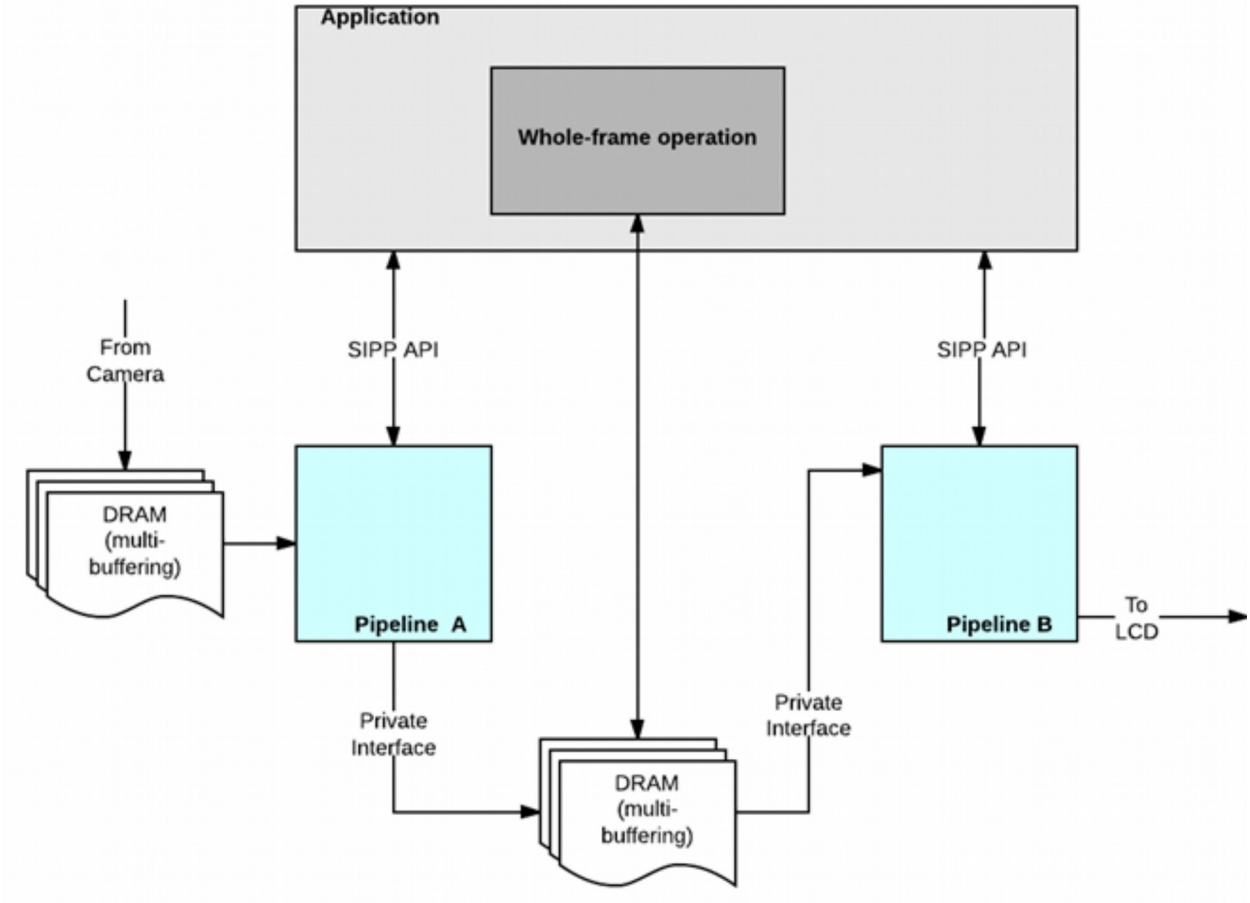

Figure 6: An application with multiple SIPP pipelines

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.

3 Memory Usage with SIPP

This section describes how SIPP uses CMX and DDR memory. Memory is used at two different places, i) at pipeline initialization and ii) 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.

Figure 7 shows how CMX is used.

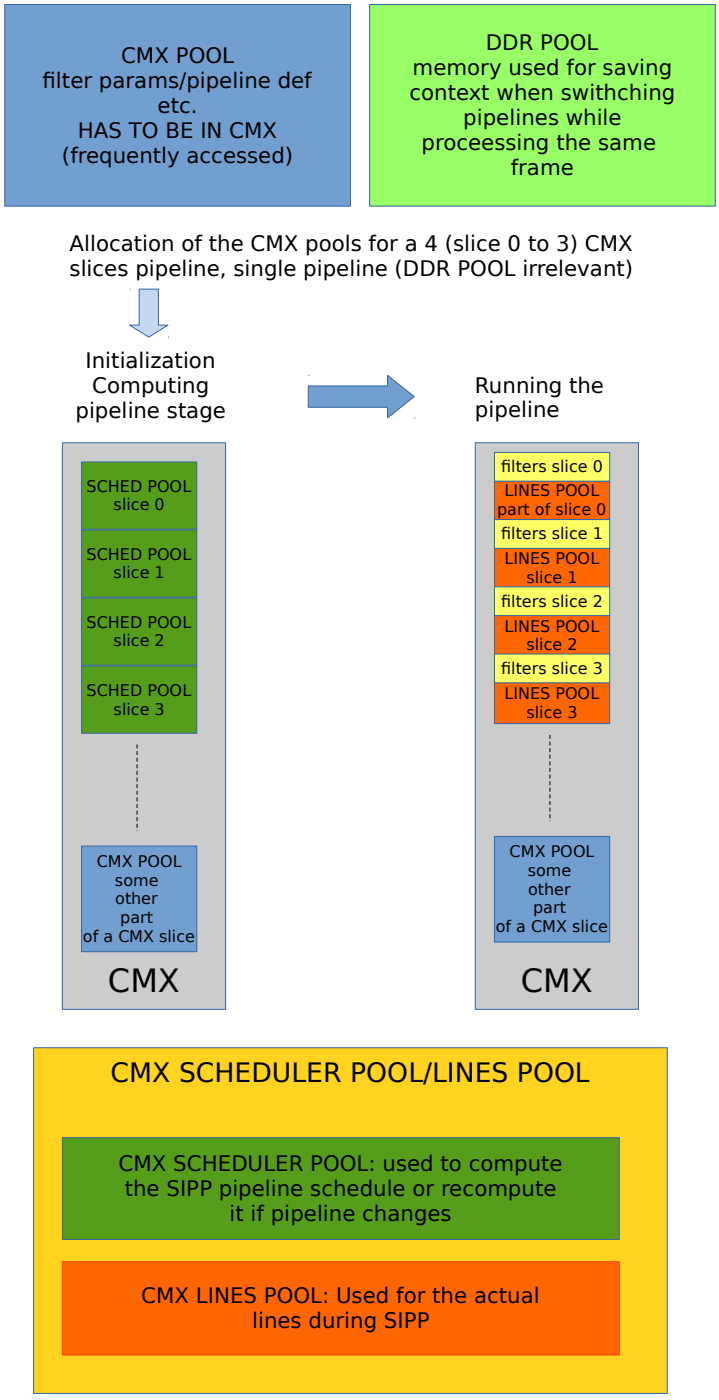

Figure 7: CMX use

To use DDR, the SIPP framework provides a function

```
sippInitSchedPoolArb(bigSchedBuf, sizeof(bigSchedBuf));
```

This allows the application developer allocate memory required to different locations. For example, this may be required if changes are required to the pipeline while the pipeline has filters code loaded in slices.

Figure 8 shows how DDR is used with SIPP applications

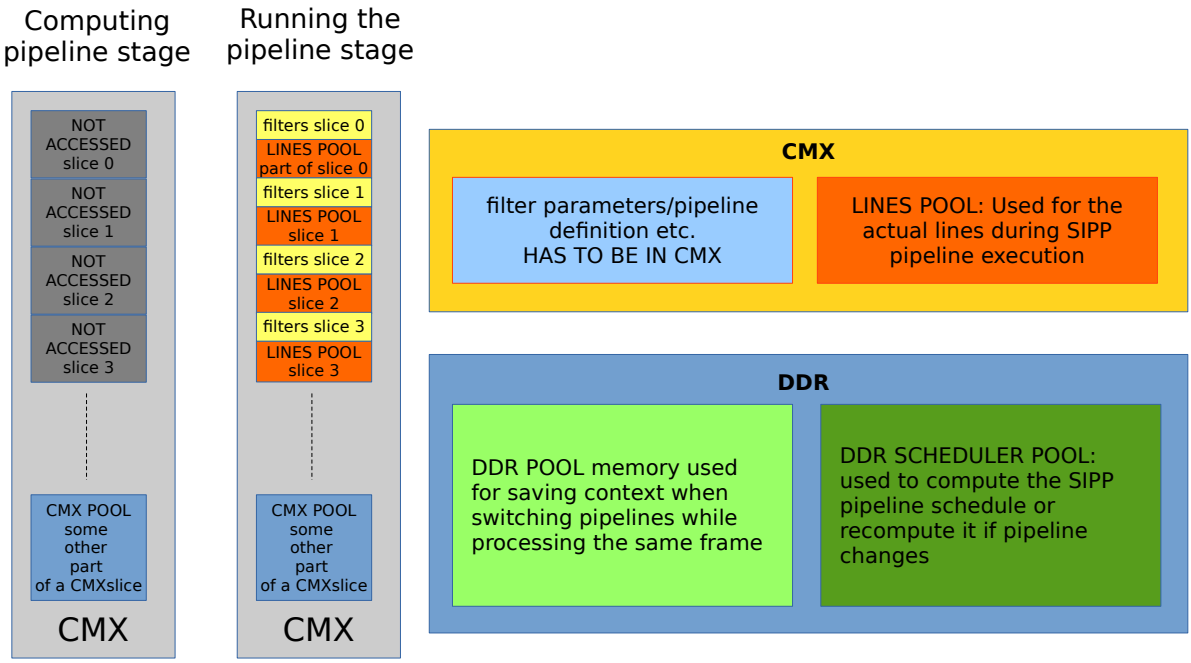

Figure 8: DDR use with SIPP applications

4 Getting started

Applications which use the SIPP framework can be developed and run on Myriad 1 or Myriad 2 SOCs, but also in a PC simulation environment.

SIPP example applications are provided in the `mdk/examples` folder. These examples are cross-compilable for different environments e.g. PC, Myriad 1 / Myriad 2 (targeting real hardware or MoviSim simulator).

4.1 Myriad 1

4.1.1 Running SIPP on the Myriad 2 Development system

Some basic SIPP sample applications are provided with the MDK (Myriad Development Kit). These are located in the `mdk/examples/myriad1/Sipp` folder. These apps are built and run in the same manner as other MDK example apps (see the "MDK getting started" document for more details).

You can use the sample apps as a starting point for more complex applications.

4.2 Myriad 2

Myriad 2 SIPP examples are located in `mdk/examples/myriad2/sipp` folder.

4.3 SIPP Simulation on PC

4.3.1 Windows PC environment

It is possible to build SIPP examples for the PC target using Microsoft VisualStudio project files for VisualStudio2012 and for Linux and for Cygwin.

The MDK also contains pre-built libraries for the required swSipp, and hwModels libraries:

mdk/common/components/sipp/prebuild

The build folder for each SIPP example contains the required files to build for the required environment.

4.3.2 Linux PC environment

The *make* system is used to build SIPP examples for the Linux environment. The SIPP examples are located in the MDK/examples folder, relevant to the target being built.

- mdk/examples/myriad1/Sipp
- mdk/examples/myriad2/Sipp
- mdk/examples/portable/Sipp

To build the examples run the command:

```
% make all
```

This *makefile* will also build the examples in the Windows Cygwin environment.

5 The SIPP API

5.1 Overview

This chapter describes the API which is used to instantiate and run processing pipelines which use the SIPP

framework. Applications may call the SIPP API in order to instantiate a pipeline, instantiate filters within the pipeline, and connect the filters together to form a SIPP pipeline graph.

5.2 API conventions

5.2.1 Errors

The API does not return error codes, as all errors are considered fatal. The SIPP implementation traps failures by using assertions. Upon an assertion failure, execution will halt, and a message will be printed to the console (if running a PC simulation) or to the UART device, if UART output is enabled (note that UART output is normal redirected to the debugger during development). If the application is being run inside a debugger, you can use the debugger to perform a stack backtrace in order to establish where the failure occurred. You do not need to check for error codes or NULL pointers being returned by SIPP API calls, because the calls will never return if a failure occurs.

5.3 sippPlatformInit()

5.3.1 Prototype

```
void sippPlatformInit();
```

5.3.2 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.

5.3.3 Parameters

-	No parameters.

5.4 sippInitialize()

5.4.1 Prototype

```
void sippInitialize();
```

5.4.2 Description

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

5.4.3 Parameters

-	No parameters.

5.5 sippCreatePipeline()

5.5.1 Prototype

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

5.5.2 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.

5.5.3 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.

5.5.4 Example

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

5.6 sippCreateFilter()

5.6.1 Prototype

```
SippFilter* sippCreateFilter(SippPipeline *pl, u32 flags,
                             u32 outW, u3outH, u32 numPl,
                             u32 bpp, u32 paramsAlloc,
                             void (*funcSvuRun)(struct SippFilterS
                                  *fptr, int svuNo, int runNo),
                             const char *name);
```

5.6.2 Description

This function instantiates a SIPP pipeline, 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).

5.6.3 Parameters

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

5.7 sippLinkFilter()

5.7.1 Prototype

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

5.7.2 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. When using this call to build the graph, you need to make sure the graph conforms to the rules and guidelines specified in section 8.

5.7.3 Parameters

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

5.8 sippFinalizePipeline()

5.8.1 Prototype

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

5.8.2 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* 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.

5.8.3 Parameters

pl	Pointer reference to the pipeline

5.9 sippProcessFrame()

5.9.1 Prototype

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

5.9.2 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.

5.9.3 Parameters

pl	Pointer reference to the pipeline to run

5.10 sippProcessIters()

5.10.1 Prototype

```
void sippProcessIters(SippPipeline *pl, UInt32 numIters);
```

5.10.2 Description

Invokes the SIPP scheduler to process numIters worth of data e.g. lines. 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. With this function, processing can be paused and restarted.

5.10.3 Parameters

pl	Pointer reference to the pipeline to run.
numIters	Number of iterations to run.

5.11 sippReschedule()

5.11.1 Prototype

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

5.11.2 Description

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

The pipeline schedule is normally calculated once at application initialization (or it can be pre-calculated by running the PC version of the pipeline application). 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().

5.11.3 Parameters

pl	Pointer reference to the pipeline

5.12 SIPP Utility Functions

5.12.1 sippInitSchedPoolArb()

5.12.1.1 Prototype

```
Void sippInitSchedPoolArb(UInt8 *addr, UInt32 size);
```

5.12.1.2 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.

5.12.1.3 Parameters

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

5.12.2 sippRdFileU8()

5.12.2.1 Prototype

```
void sippRdFileU8(UInt8 *buff, int count, const char *fName);
```

5.12.2.2 Description

Read a file containing unsigned 8bit integers.

5.12.2.3 Parameters

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

5.12.3 sippWrFileU8()

5.12.3.1 Prototype

```
void sippWrFileU8(UInt8 *buff, int count, const char *fName);
```

5.12.3.2 Description

Write unsigned 8bit integers to a file.

5.12.3.3 Parameters

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

5.12.4 SippRdFileU8toF16()

5.12.4.1 Prototype

```
void sippRdFileU8toF16(UInt8 *buff, int count, const char *fName);
```

5.12.4.2 Description

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

5.12.4.3 Parameters

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

5.12.5 sippWrFileF16toU8()

5.12.5.1 Prototype

```
void sippWrFileF16toU8(UInt8 *buff, int count, const char *fName);
```

5.12.5.2 Description

Write a buffer of 16bit floats to a file as 8bit integers.

5.12.5.3 Parameters

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

5.12.6 sippDbgCompareU8()

5.12.6.1 Prototype

```
void sippDbgCompareU8(UInt8 *refA, UInt8 *refB, int len);
```

5.12.6.2 Description

Compare two unsigned 8bit integer (char) buffers.

5.12.6.3 Parameters

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

5.12.7 sippDbgCompareU16()

5.12.7.1 Prototype

```
void sippDbgCompareU16(UInt16 *refA, UInt16 *refB, int len);
```

5.12.7.2 Description

Compare two unsigned 16bit buffers values

5.12.7.3 Parameters

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

5.12.8 sippDbgCompareU32()

5.12.8.1 Prototype

```
void sippDbgCompareU32(UInt32 *refA, UInt32 *refB, int len);
```

5.12.8.2 Description

Compare two unsigned 32bit unsigned integer buffers.

5.12.8.3 Parameters

refA	First buffer

refB	2nd buffer for comparison
len	Number of bytes to compare

5.12.9 sippUtilComputeFp16Lut()

5.12.9.1 Prototype

```
void sippUntilComputeFp16Lut(half (*formula)(half input),
                             half *outLut,
                             UInt32 lutSize);
```

5.12.9.2 Description

Fills a Look-up-table of size lutSize using the function, passed as a parameter.

5.12.9.3 Parameters

formula	Pointer to a function taking half (fp16) type parameter and returning half type
outLut	Pointer to look-up-table memory
lutSize	Size of look-up-table

5.12.10 sippUtilComputeFp16Lut()

5.12.10.1 Prototype

```
void sippUtilPrintFp16Lut(half *fp16Lut,
                          UInt32 lutSize,
                          const char *fName)
```

5.12.10.2 Description

This function is only available when running SIPP in PC mode i.e. not on chip or movisim simulator.

The function prints a C definition for the look-up-table held in memory at fp16Lut address. This can then be included in C code.

Used in conjunctions with *sippUntilComputeFp16Lut*

5.12.10.3 Parameters

fp16Lut	Pointer to look-up-table memory
lutSize	Size of look-up-table
fName	Name of file where C data-structure prototype will be written

5.13 sippAllocCmxMemRegion

5.13.1 Prototype

5.13.2 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.

5.13.3 Parameters

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

5.13.4 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 128kB) 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()

5.13.5 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 128kB 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 128kB 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 128kB, HW constraints on the slice stride must be adhered to. A single global slice size is programmable. The minimum size is 32kB and the maximum is 480kB, programmable in increments of 32kB.

6 Using the SIPP framework

6.1 Building a SIPP application

The simplest way to create SIPP pipeline application is to use the Graph Designer [6]. This is a plug-in to the moviEclipse IDE that allows users create, build and execute applications from the IDE.

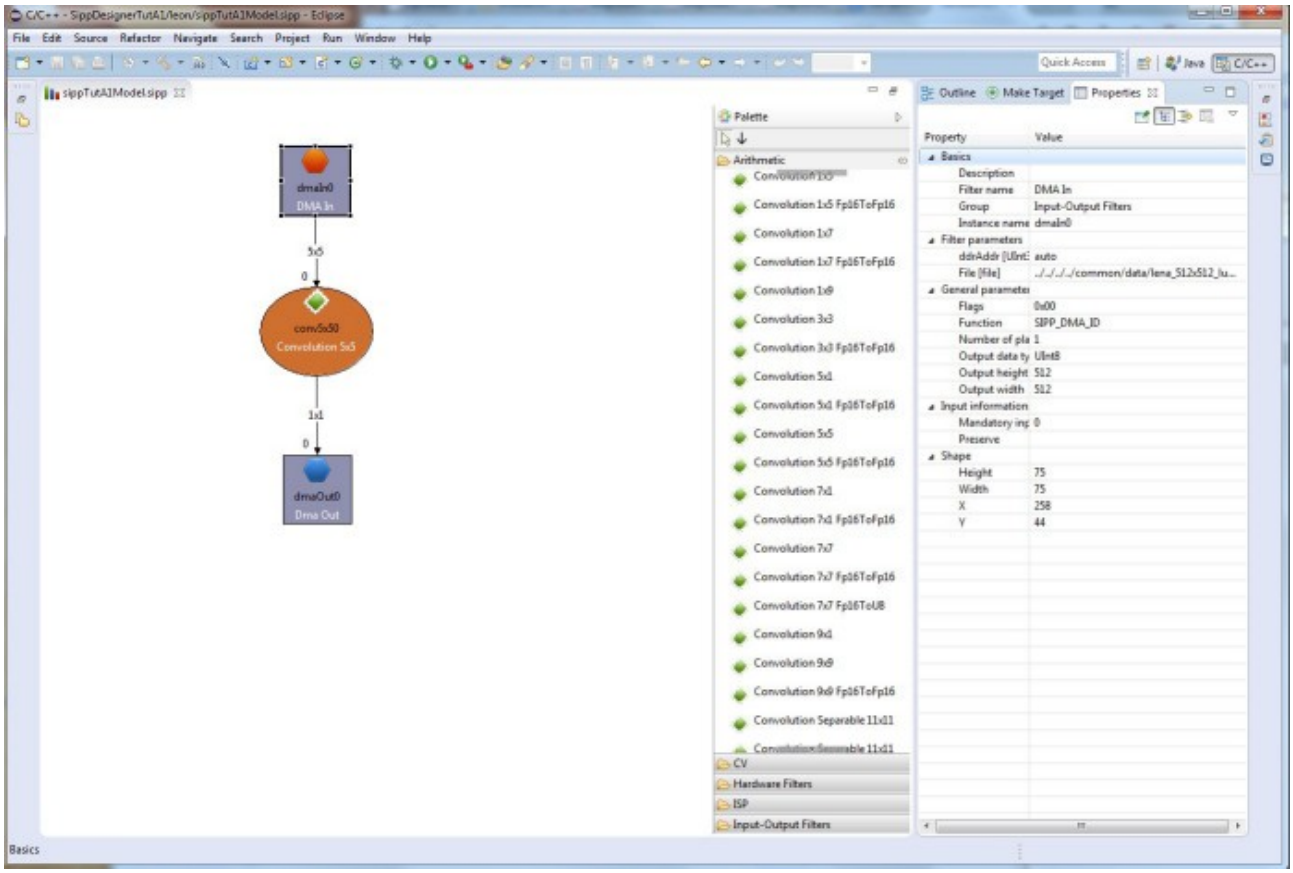

Figure 9: Graph Designer Eclipse Plug-in

The remaining sections describe internal details 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 the following steps:

- 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().*

NOTE: In a future release, a graphical tool will be provided to assist with generating the graph. This tool generates C code to instantiate the specified pipeline. Filters will be bundled with attribute files which will be utilized by the graph generation tool.

6.2 Configuring filters

Filters may have zero or more configurable parameters. The parameters are specified via a filter-specific parameter structure. The size of the configuration structure, in bytes, must be specified via the *paramsAlloc* parameter to *sippCreateFilter()*.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;
```

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

6.3 Pipeline Examples

6.3.1 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 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 (these functions only work on the PC version. When run on hardware, or in MoviSim, they have no effect).

```
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;
}
```

6.3.2 Two Pipeline processed in Iterations

In this example we show how to build a SIPP pipeline with a single filter. This example shows how the SIPP

allows context switching between two pipelines.

Build the pipeline-A (a simple DMA in, Convolution, DMA out pipeline)

```
void appSetupPipelineA()
{
   plA        = sippCreatePipeline(2, 2, SIPP_MBIN(mbinImgSipp));
   plA->flags |= PLF_RUNS_ITER_GROUPS;

   //Define the filters
   plA_dmaIn  = sippCreateFilter(plA, 0x00, IMG_W, IMG_H,
                                 N_PL(1), SZ(UInt8),
                                 SIPP_AUTO,
                                 (FnSvuRun)SIPP_DMA_ID,
                                 0);
   plA_conv   = sippCreateFilter(plA, 0x00, IMG_W, IMG_H,
                                 N_PL(1), SZ(UInt8),
                                 SIPP_AUTO,
                                 (FnSvuRun)SIPP_CONV_ID,
                                 0);
   plA_dmaOut = sippCreateFilter(plA, 0x00, IMG_W, IMG_H,
                                 N_PL(1), SZ(UInt8),
                                 SIPP_AUTO,
                                 (FnSvuRun)SIPP_DMA_ID,
                                 0);
   //Link the filters
   sippLinkFilter(plA_conv,   plA_dmaIn,  KER_SZ,  KER_SZ);
   sippLinkFilter(plA_dmaOut, plA_conv,   1,       1);
}
```

In the function to build pipeline-B,

```
void appSetupPipelineB()
{
   plB       = sippCreatePipeline(2, 2, SIPP_MBIN(mbinImgSipp));
   plB->flags |= PLF_RUNS_ITER_GROUPS;

   //Filters
   plB_dmaIn  = sippCreateFilter(plB, 0x00, IMG_W, IMG_H,
                                 N_PL(1), SZ(UInt8),
                                 SIPP_AUTO,
                                 (FnSvuRun)SIPP_DMA_ID,
                                 0);
   plB_conv   = sippCreateFilter(plB, 0x00, IMG_W, IMG_H,
                                 N_PL(1), SZ(UInt8),
                                 SIPP_AUTO,
                                 (FnSvuRun)SIPP_CONV_ID,
                                 0);
   plB_dmaOut = sippCreateFilter(plB, 0x00, IMG_W, IMG_H,
                                 N_PL(1), SZ(UInt8),
                                 SIPP_AUTO,
                                 (FnSvuRun)SIPP_DMA_ID,
                                 0);
     //Links
   sippLinkFilter(plB_conv,   plB_dmaIn,  KER_SZ,  KER_SZ);
   sippLinkFilter(plB_dmaOut, plB_conv,   1,       1);
}
```

We create and link as a normal pipeline but we configure it to indicate that it will be executed by iterating between the pipelines, `flags |= PLF_RUNS_ITER_GROUPS;`

In the main application, we initialize the Myriad chip and the SIPP framework, Create the pipelines and we finalize the schedules after the schedules for each individual pipeline has been created.

Execution is now under under the control and we show each separate iteration and the switching between pipelines:

```
int main()
{
    sippPlatformInit();

    appSetupPipelineA();
    appSetupPipelineB();

    sippFinalizePipeline(plA);
    sippFinalizePipeline(plB);

    // Interleaved iter groups, total iterations = 13,
    sippProcessIters(plA, 2);
    sippProcessIters(plB, 4);
    sippProcessIters(plA, 3);
    sippProcessIters(plB, 5);
    sippProcessIters(plA, 1);
    sippProcessIters(plB, 4);
    sippProcessIters(plA, 7);
}
```

6.4 Configuring SIPP

6.4.1 Run-time Execution

6.4.1.1 CMX Pool Size

The user can control the size of SIPP CMX-memory pool. The default size is 32KB (as defined in sipp/build/sippMyriad2Elf.mk). If no value is defined, the size defaults to 1MB (as per /sip/core/sippMem.c). This is typically useful for PC development.

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

CCOPT += -D'SIPP_CMX_POOL_SZ=32768'

6.4.1.2 Mutex

For Shave synchronization purposes, SIPP component uses 2 mutexes. Immediately after pipeline creation (via sippCreatePipeline), these the 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;
```

6.4.1.3 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'

6.4.2 Runtime Schedule Calculation

The SIPP pipeline application is dependent on a schedule which pushes data through each kernel. The SIPP engine can calculate this schedule at runtime or it can be calculated off-line by the PC simulation and included in the pipeline application.

The SIPP PC simulation generates a file, pcDumpSchedule.hh, this can be included as follows:

```
#include "pcDumpSchedule.hh"
```

and calling:

```
dbgSchedInit(pl);
```

in the pipeline control code.

Alternatively, the schedule can be calculated dynamically at runtime with a small performance overhead at application initialization.

After the pipeline is created, call the function:

```
sippFinalizePipeline(pl);
```

At runtime the SIPP application may exit with an out-of-memory error. This is due to the SIPP not having enough memory to calculate the runtime schedule usually due to large / complex pipelines. 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 to use as a temporary workspace.

6.4.3 SIPP Build / make Parameters

The SIPP uses a number of build / make options:

- SIPP_VCS – To build for Myriad 2 chip testing mode
- SIPP_PC – To build for execution on PC (x86)
- SIPP_F_DUMPS – For PC execution only. Used to control dumping output from kernels to a file.
- SIPP_USE_PTR_GET_FUNCS – INTERNAL - DO NOT USE / MODIFY
- SIPP_CONCURRENT_MODE – Experimental – DO NOT USE / MODIFY
- SIPP_ITER_PROF_TIM_BASE – INTERNAL – DO NOT USE / MODIFY
- OPT_EXE_NO_UPD – INTERNAL – DO NOT USE / MODIFY

7 SIPP Hardware Accelerator Filters

With the exception of the DMA filter, the Hardware filters defined in this section are for Myriad 2 pipelines only.

The Hardware filters defined in this section are for Myriad 2 pipelines only

For each hardware-accelerated filter, there is a software filter which "drives" the hardware. From an application programmer's point of view, using a hardware filter is no different from using a software filter. Each hardware filter may have some parameters which can be configured. The software driver filter uses these parameters to program the hardware registers to configure the filter.

It is recommended that this section is read in conjunction with the Myriad 2 datasheet [4].

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 to produce a single line 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
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
Sharpen	SIPP_SHARPEN_ID	U8, FP16	U8, FP16	UsmParam	7
Luma Denoise	SIPP_LUMA_ID	U8, FP16	U8, FP16(Luma) U8 (Reference)	YDnsParam	11
Chroma Denoise	SIPP_CHROMA_ID	U8	U8	ChrDnsParam	11
Median	SIPP_MED_ID	U8	U8	MedParam	3/5/7
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	9
Convolution	SIPP_CONV_ID	U8, FP16	U8, FP16	ConvParam	3/5

Filter	Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
Color combination	SIPP_CC_ID	FP16,U8 (luma), U8 (chroma)	FP16/U8 (RGB)	ColCombParam	1 (Lum), 4 (Chr)
Debayer post-processing	SIPP_DBYR_PPM_ID	U8, U16	U8, U16	DbyrPpParam	

Table 3: Summary of Myriad 2 Hardware Filters

Filter ID	Filter	Buffer	Input buffer ID	Output buffer ID
		Primary buffers		
0	RAW	in/out	0	0
1	LSC	in/out	1	1
2	Debayer	Bayer in/RGB out	2	2
3	Chroma denoise	in/out	3	3
4	Luma denoise	in/out	4	4
5	Sharpen	in/out	5	5
6	Polyphase scaler	in/out	6	6
7	Median	in/out	7	7
8	Look-up table	in/out	8	8
9	Edge operator	in/out	9	9
10	Convolution kernel	in/out	10	10
11	Harris corners	in/out	11	11
12	Color combination	Luma in/RGB out	12	12
13	Reserved	–	13	13
14	MIPI Tx[0]	in	14	–
15	MIPI Tx[1]	in	15	–
16	MIPI Rx[0]	out	N.A.	16
17	MIPI Rx[1]	out	N.A.	17
18	MIPI Rx[2]	out	N.A.	18
19	MIPI Rx[3]	out	N.A.	19
20	Debayer post-proc median	in	20	20
		Secondary buffers		
	RAW	Stats out	N.A.	15
	LSC	Gain mesh in	16	N.A.
	Chroma denoise	Reference in	17	N.A.
	Luma denoise	Reference in	18	N.A.
	Look-up table	Look-up table	19	N.A.
	Color combination	Chroma in	21	N.A.

Table 4: SIPP filter IDs and input/output buffer IDs

7.1 Hardware Filter Throughput and Performance

Generally the filter data-paths are designed for a throughput of one clock cycle per pixel (1cc/pixel). However, in certain configurations, the throughput of filters using large kernels may be limited by the

memory read bandwidth required.

Consider a filter using a 7x7 pixel kernel processing an FP16 buffer (2 bytes per pixel). The filter must read data from 7 consecutive lines in its input buffer to fill each column of its pixel kernel. If the filter has only one AMC read client interface (64 bit) then only 4 FP16 pixels can be read per clock cycle. But if the filter were to sustain a throughput of one clock cycle per pixel, it would have to be able to read 7 pixels per clock cycle. Assuming the filter is able to receive data on every clock cycle the actual throughput in this case is 7/4 = 1.75cc/pixel. This number would be the maximum theoretical throughput achievable for the filter (in the given configuration).

In practice attained throughput may be less than the theoretical maximum due to contention with other filters for Slice access in the AMC and/or memory cut clashes within the Slice. Filter buffers should be laid out in CMX with care to avoid such scenarios if possible. (Slice chunked mode is useful in this regard.)

Table 5 outlines the maximum theoretical throughput for a range of pixel kernels and data types, assuming a single 64 bit read client interface to the AMC.

Kernel lines	Bytes per pixel	Max throughput
1	2	1cc/pixel
2	2	1cc/pixel
3	2	1cc/pixel
5	2	1.25cc/pixel
7	2	1.75cc/pixel
9	2	2.25cc/pixel
11	2	2.75cc/pixel
1	1	1cc/pixel
2	1	1cc/pixel
3	1	1cc/pixel
5	1	1cc/pixel
7	1	1cc/pixel
9	1	1.125cc/pixel
11	1	1.375cc/pixel
15	1	1.875cc/pixel

Table 5: Filter throughput given a single 64 bit AMC read client interface

NOTE: Filters generally require much less memory write bandwidth than read and that, generally, if a filter's output buffer (or chunk thereof) is allocated in a different CMX Slice to its input buffer then contention between the read and write clients can be avoided and throughput will not be limited by write bandwidth.

In most cases then, available read bandwidth is the determining factor for throughput. Therefore, the majority of filters have been allocated a sufficient number of read client interfaces to allow a throughput of 1cc/pixel to be sustained (assuming no stalling), even in their maximal configuration (in terms of kernel lines and bytes per pixel). Table 6 outlines the read client allocation for each filter and its maximum throughput

(rounded up to 1 decimal place) for its maximal configuration. Filters unable to sustain a throughput of 1cc/pixel are highlighted. Of these, chroma denoise is assumed to operate on image planes sub-sampled horizontally and vertically by a factor of 2 (relative to those processed by other filters). Its throughput relative to filters processing full size images need therefore only be < 4cc/pixel. The other filters highlighted have sub-maximal configurations (using smaller pixel kernels or fewer bytes per pixel) at which 1cc/pixel is attainable.

Filter ID	Filter	Buffer	Maximal config	Read clients	Max throughput
		Primary buffers			
0	RAW	in/out	5 lines, 2 bytes per pixel	2	1cc/pixel
1	LSC	in/out	1 line, 2 bytes per pixel	1	1cc/pixel
2	Debayer	Bayer in/RGB out	11 lines, 2 bytes per pixel	2	1cc/pixel
3	Chroma denoise	in/out	21 lines, 1 byte per pixel	1 per plane	2.7cc/pixel
4	Luma denoise	in/out	7 lines, 2 bytes per pixel	2	1cc/pixel
5	Sharpen	in/out	7 lines, 2 bytes per pixel	2	1cc/pixel
6	Polyphase scaler	in/out	7 lines, 2 bytes per pixel	1	1.8cc/pixel
7	Median	in/out	7 lines, 1 byte per pixel	1	1cc/pixel
8	Look-up table	in/out	4 planes, 2 bytes per pixel	1	1cc/pixel
9	Edge operator	in/out	3 lines, 2 bytes per pixel	1	1cc/pixel
10	Convolution kernel	in/out	5 lines, 2 bytes per pixel	2	1cc/pixel
11	Harris corners	in/out	9 lines, 1 byte per pixel	1	1.2cc/pixel
12	Color combination	Luma in/RGB out	1 line, 2 bytes per pixel	1	1cc/pixel
13	Reserved	–	–	–	–
14	MIPI Tx[0]	in	1 line, 4 bytes per pixel	1	1cc/pixel
15	MIPI Tx[1]	In	1 line, 4 bytes per pixel	1	1cc/pixel
16	MIPI Rx[0]	out	1 line, 4 bytes per pixel	–	1cc/pixel
17	MIPI Rx[1]	out	1 line, 4 bytes per pixel	–	1cc/pixel
18	MIPI Rx[2]	out	1 line, 4 bytes per pixel	–	1cc/pixel
19	MIPI Rx[3]	out	1 line, 4 bytes per pixel	–	1cc/pixel
20	Debayer post-proc median	in	3x planes of 3 lines, 2 bytes per pixel	1 per plane	1cc/pixel
		Secondary buffers			
	RAW	Stats out	N.A.	–	–
	LSC	Gain mesh in	2 lines, 2 bytes per sample	1	1cc/pixel

Filter ID	Filter	Buffer	Maximal config	Read clients	Max throughput
	Chroma denoise	Reference in	21 lines, 1 byte per pixel	1 per plane	2.7cc/pixel
	Luma denoise	Reference in	11 lines, 1 byte per pixel	2	1cc/pixel
	Look-up table	Look-up table	N/A	–	–
	Color combination	Chroma in	3x planes of 5 lines, 1 byte per pixel	1 per plane	1cc/pixel

Table 6: Filter read client allocation and maximum throughput in maximal config

7.2 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).

For Myriad 2 the full description of the DMA and its usage is contained in the Myriad 2 datasheet [4].

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.

7.2.1 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.

7.2.2 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.

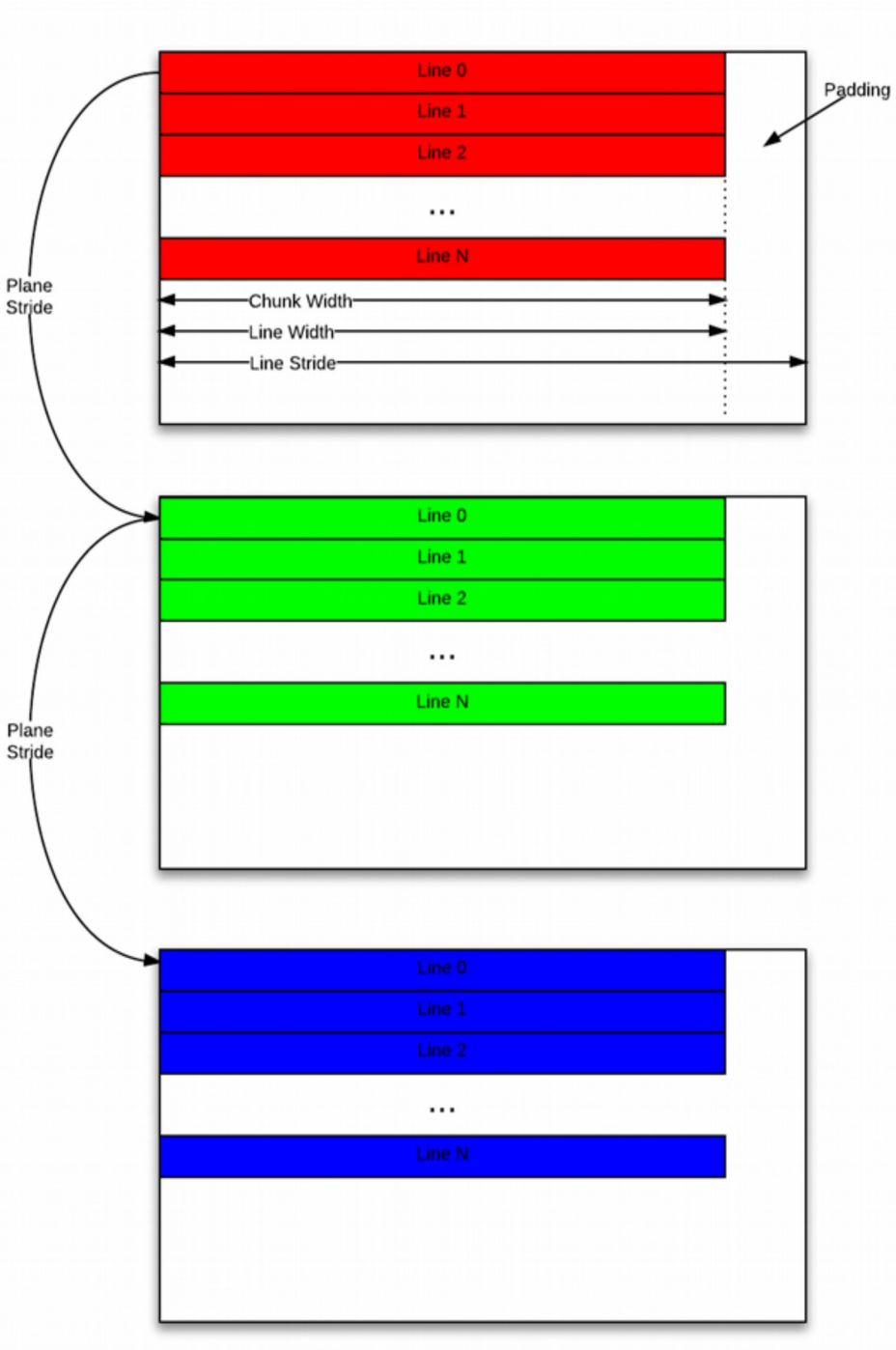

Figure 10: DMA configuration targeting image in DDR

7.2.3 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.

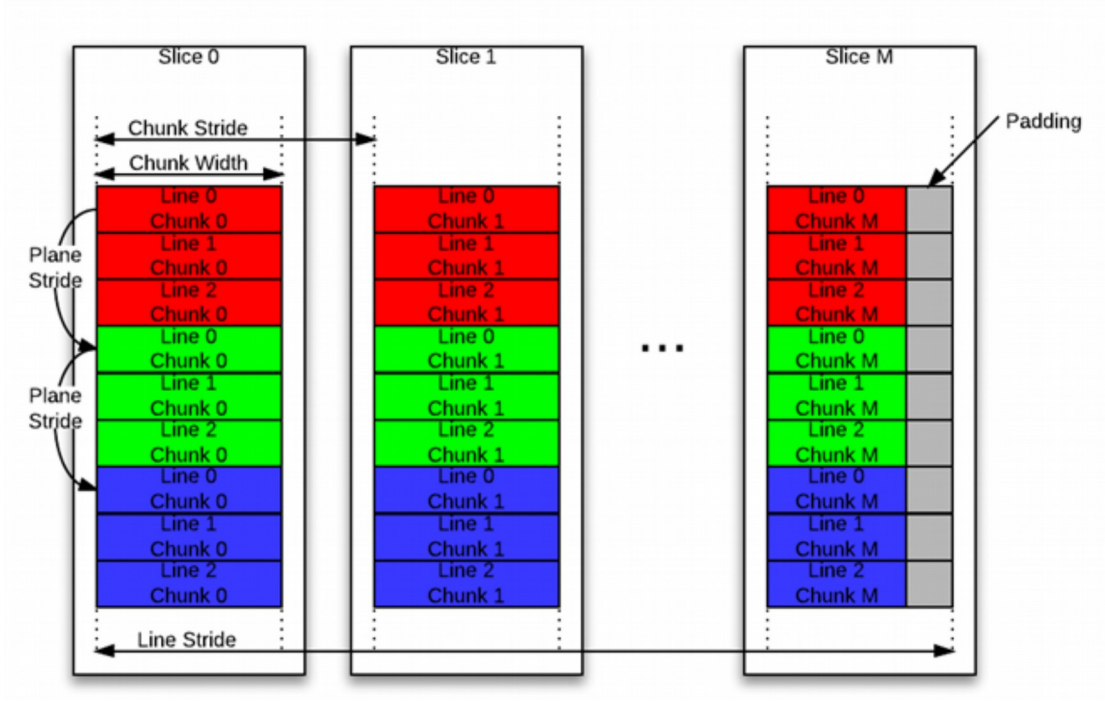

Figure 11: DMA configuration targeting sliced image in CMX memory

7.2.4 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

7.3 MIPI Rx

Filter ID	Output Precision	Output Precision	Config Struct	Input Lines
SIPP_MIPI_RX0_ID SIPP_MIPI_RX1_ID SIPP_MIPI_RX2_ID SIPP_MIPI_RX3_ID	U8, U16, FP16, 10P32	N/A	MipiRxParam	N/A
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 7: 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 8) 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 8: 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 12 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.

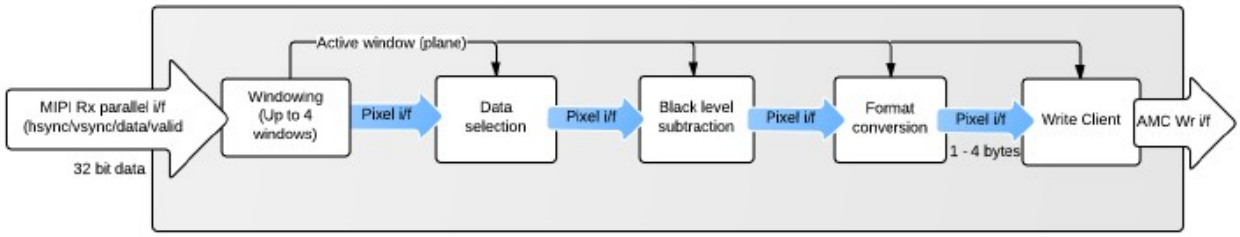

Figure 12: MIPI Rx filter block diagram

7.3.1 MIPI Rx Features

7.3.1.1 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:

R G
B W

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 13 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.

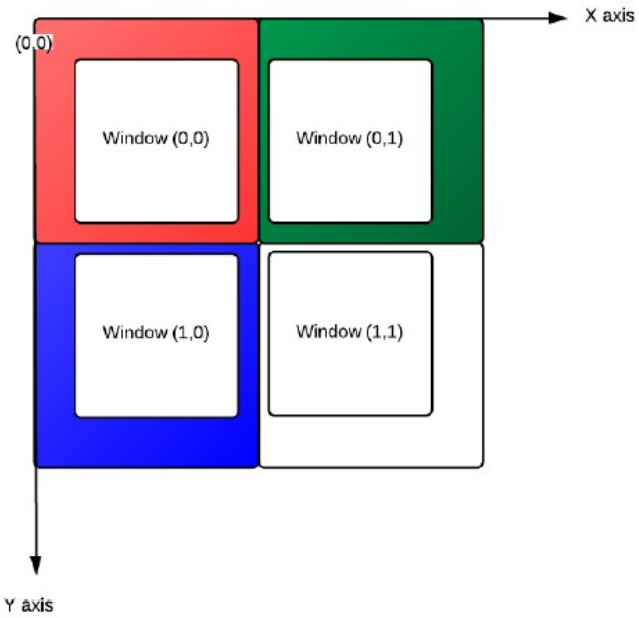

Figure 13: Example MIPI Rx window grid for RAW data from RGBW array sensor

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.

7.3.1.2 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.

7.3.1.3 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.

7.3.1.4 Format conversion

In 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.

7.3.1.5 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.

7.3.1.6 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.

7.3.2 MIPI Rx Configuration

Name	Bits	Description
frmDim	 15:0 31:16	PRIVATE - frame dimensions in pixels Frame width Frame height
cfg	1:0	Clock Speed

Name	Bits	Description
		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) 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	Least significant bit of Window 2 selection

Name	Bits	Description
	11:8 19:15 27:24	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.

7.4 MIPI Tx

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_MIPI_TX0_ID SIPP_MIPI_TX1_ID	U8, U16, U24, U32 10P32	N/A	MipiTxParam	N/A
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 9: MIPI *Tx* filter overview

7.4.1 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
	7:6	Vertical interval in which to generate vertical interval interrupt 00 – VSYNC 01 – Back Porch 10 – Active 11 – Front Porch
	8	Level of VSYNC when timing FSM is in BACKPORCH state
	9	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

Name	Bits	Description
		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
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

7.5 Lens Shading Correction (LSC)

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_LSC_ID	U8, U16 (Image), U8.8 (Mesh)	U8, U16	LscParam	1

Input	Up to 16 planes (sequentially) of up to 16 bit RAW (Bayer pattern or non Bayer data)
Operation	Lens shading/color shading correction via application of per pixel gains interpolated from sub-sampled gain-mesh
Input buffers	Minimum of 1lines for data (image) buffer Minimum dimensions of gain mesh 2x2 (bi-linear interpolation kernel is 2x2), maximum 1023x1023 (but gain mesh must not exceed dimensions of input image)
Output	Up to 16 planes of up to 16 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 centre 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.

7.5.1 LSC Features

7.5.1.1 Correction Mesh

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.

7.5.2 LscParam Configuration

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

Name	Bits	Description
gmBase	*Uint16	CMX memory address of the gain correction mesh
gmWidth	9:0	Width of the gain mesh – must be a multiple of 4
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.

7.6 RAW filter

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_RAW_ID	U8, U16	U8, U16	RawParam	1/3/5

Input	Up to 16 planes (sequentially) of up to 16 bit RAW (Bayer pattern or non Bayer data)
Operation	Gr/Gb imbalance correction Hot pixel suppression Digital gain and saturation Patched based color channel/plane accumulation statistics histogram
Input buffer	Minimum of 5 lines if Gr/Gb imbalance or hot pixel suppression is enabled, minimum of 3 lines if histogram is enabled for Bayer data, otherwise a minimum of 1 line.
Output	Up to 16 planes (sequentially) of up to 16 bit RAW (Bayer pattern or non Bayer data) + accumulation statistics.
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. The Raw filter performs a number of functions on raw CFA data, prior to demosaicing, including hot and cold pixel suppression, Gr/Gb imbalance correction, digital gain, and statistics collection.

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.

Accumulation and histogram statistics gathering may optionally be gathered on a luma channel derived from input Bayer data or on the designated white or clear channel (plane) for non-Bayer. Note that the uncorrected input data is used for the gathering of statistics.

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.

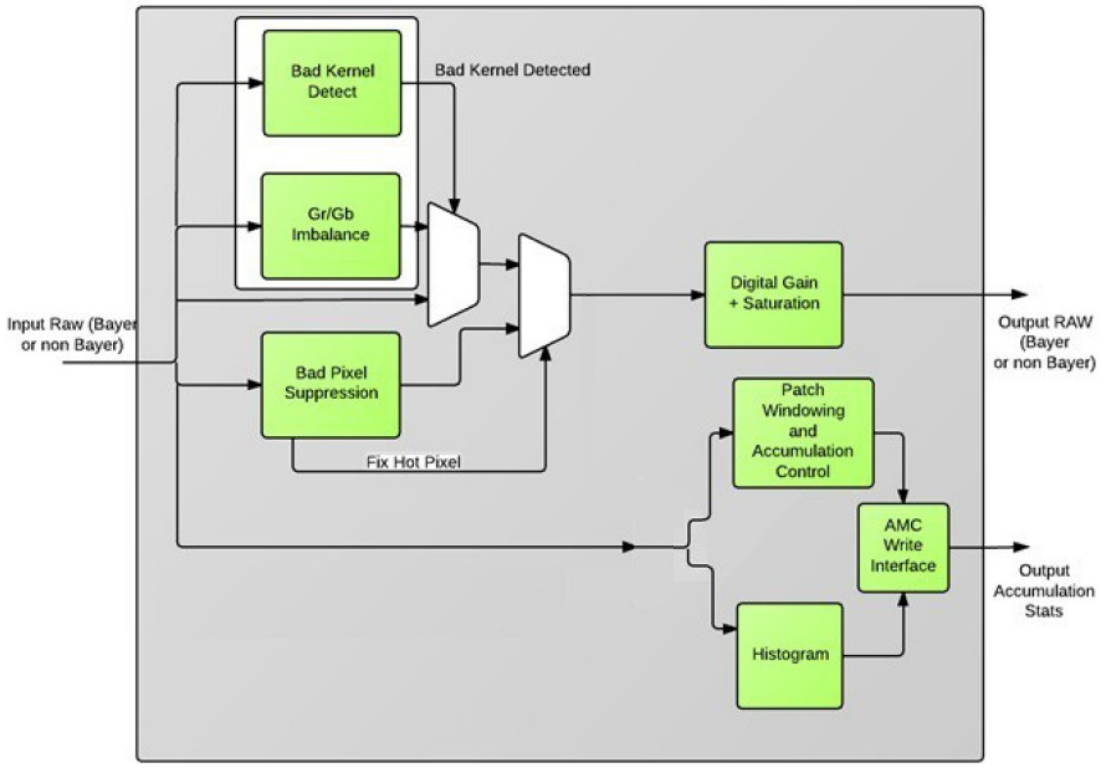

Figure 14: Raw Block

7.6.1 Features

7.6.1.1 Gr/Gb Imbalance

This filter computes a local difference between each green pixel, and its four diagonal neighbors. Then a function is applied to the green difference, and the result is used to reduce the local difference, while minimizing suppression of detail. There are configurable thresholds (“plat”, short for Plateau, and “decay”) which must be programmed to control the differentiation between detail vs. Local Gr/Gb channel imbalance. The thresholds must be tuned depending on the camera sensor, and may be configured independently for dark vs. bright areas of the image (the dark and bright thresholds are interpolated, according to the local pixel intensity).

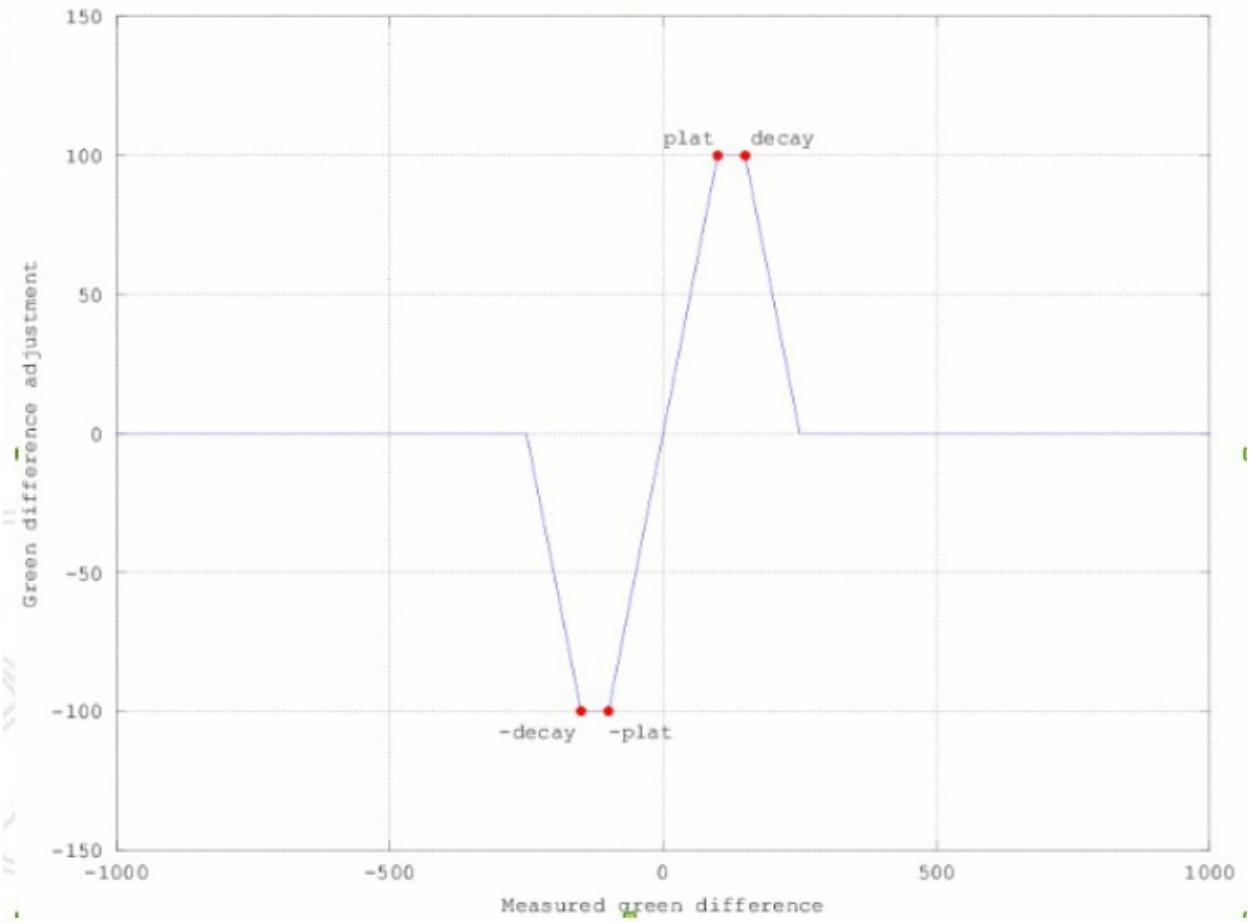

Figure 15: Function applied to local green difference in Gr/Gb imbalance filter, showing programmable thresholds "plat" and "decay"

The Gr/Gb imbalance filters works in the range [0, 4095]. Input values are scaled into this range, and all programmable thresholds are also specified within this range. The "plat" threshold sets the maximum correction that can be applied by the filter. If the absolute value of the local difference is greater than ("decay" + "plat"), then no correction will be applied. Note that the "decay" value should be >= the corresponding "plat" value.

7.6.1.2 Hot/Cold pixel suppression

Hot and cold pixels are detected by analyzing the local gradients in the pixel neighborhood. Programmable controls are used to control the aggressiveness of the filter. These thresholds are specified independently for green pixels vs. Red/Blue pixels, and for Hot pixels vs. Cold pixels.

This filter can be configured to only operate on green pixels, leaving Red and Blue pixels alone. This can be useful for processing some types of non-Bayer CFA images.

7.6.1.3 Histogram

A histogram of the luma channel is gathered across the frame on the programmed plane. The plane for which the statistics are gathered is configured via the *statsPlanes* field. 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:

[1/16, 2/16, 1/16],

[2/16, 4/16, 2/16],

[1/16, 2/16, 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.

The histogram has 64 bins numbered 0 to 63, each containing a U32 count. Bins are indexed using the 6 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 6 most significant bits are 0x1b (decimal 27): The count in bin 27 is incremented by 1.

At the end of the frame the contents of the histogram are uploaded to statistics output buffer starting at bin 0. 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 1 plane of 1 complete frame.

7.6.2 Statistics

Statistics are computed for Auto White Balance algorithms in a patch-based manner. A programmable number of patches, of configurable size, are distributed across the image field of view. The (X, Y) location of the top-left pixel of the first patch is programmable. The number of patches is independently configurable up to a maximum of 64 in both dimensions. The patch size is independently configurable up to a maximum of 256 in both dimensions. The patch width and height must be multiples of 2.

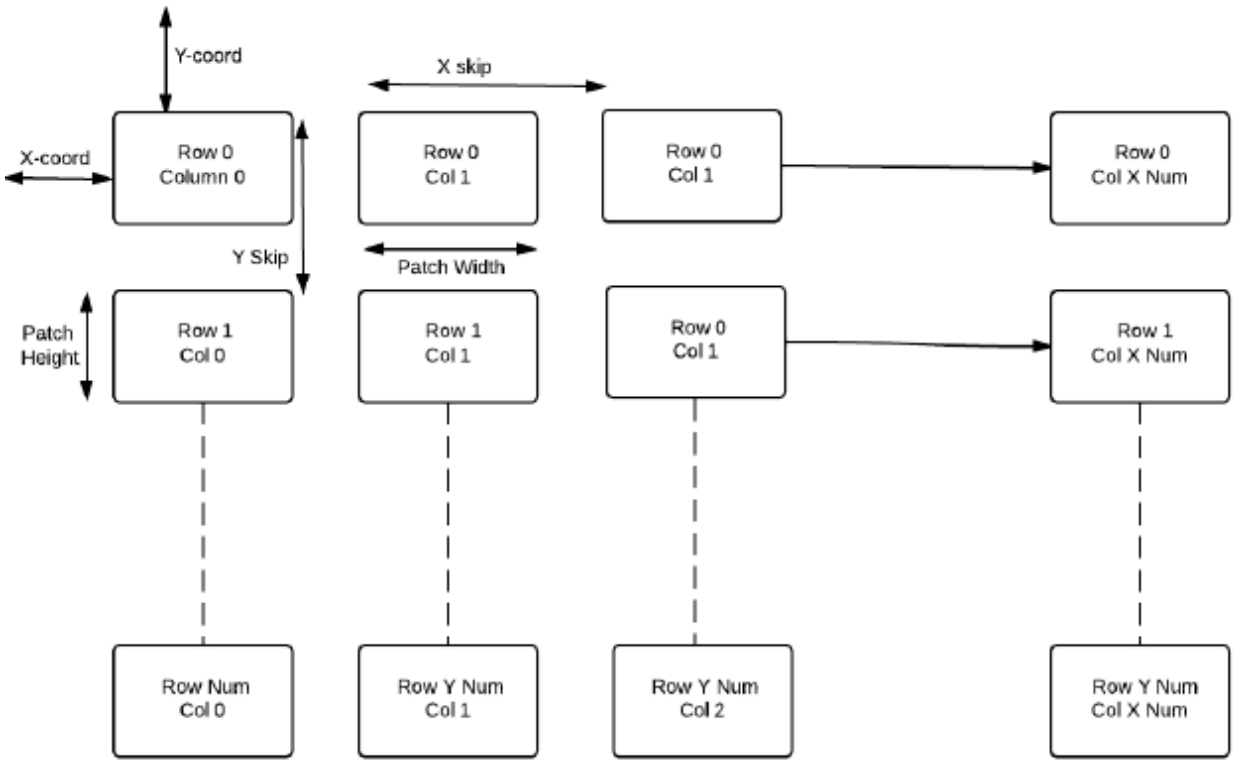

Figure 16: Patch Array

For each patch, the sum of all values in the patch is accumulated independently for each color channel.

The results are written out to memory. For each patch, 4, 32-bit words are written to memory. The patch statistics are written in raster order.

Only one plane of Bayer data can be monitored. For non Bayer data up to four planes may be monitored, also programmed separately.

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.)

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.

A local RAM is used to store the partial accumulations as lines scan through the filter. Since the patches don't overlap the hardware will only ever be working on one patch at a time. As there may be up to 64 patches on a line and up to 4 planes enabled, the hardware must maintain partial accumulations for up to 256 different patches at any one time. (This corresponds to the maximum number of patches in any one row.) At the start of a patch the relevant accumulation(s) are read out of local RAM into the accumulation

hardware. At the end of a patch in the horizontal direction (with the exception of the last line of the patch) the relevant accumulation(s) are written back to local RAM.

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.

Statistics are written to the statistics output buffer as an array of U32 values, patch accumulations first, followed by the histogram. The size of the array is equal to the total number of patches multiplied by the number of active planes plus the number of bins in the histogram (64).

Within the patch statistics the order will be the first to last patch data in the first row of patches in the lowest order enabled plane followed 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.

The histogram is appended to the patch accumulations starting from bin zero and finishing on bin 63. The histogram and patch accumulations may be enabled separately.

Figure 17 shows the layout of generated AWB statistics, when the RAW block is configured for GRBG data. Each square represents the sum of all pixels for a given channel in a given patch, stored as a 32-bit word.

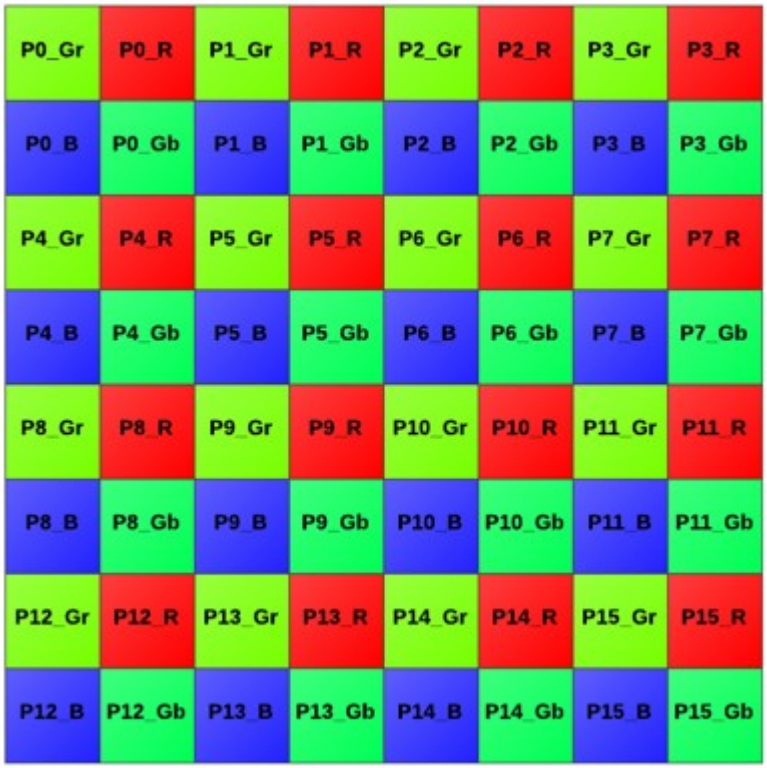

Figure 17: Layout of generated AWB statistics, for RAW GRBG data

7.6.2.1 Gain and clamp

Prior to output, an independently programmed gain value is applied to each color channel (Gr, R, B and GB). The output is then clamped. For every programmable gain value, there is a corresponding clamp value. The gains applied are normally part of the white balancing process. However, the gains and saturation values can also be used to “promote” the Bayer data to a higher bit depth. All of the other functions in the Raw filter operate at the **input** bit depth. However, the gain and clamp, which is the final stage of the Raw filter,

can be used to promote the output to a higher bit depth.

7.6.3 RawParam Configuration

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

Name	Bits	Description
FrmDim		PRIVATE – frame dimensions in pixels
	15:0	Frame width
	31:16	Frame height
grgbPlat	13:0	Plato Dark – Maximum local green difference reduction in dark areas of the image
	29:16	Plato Bright – Maximum local green difference reduction in bright areas of the image
grgbDecay	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]).
	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]).
badPixCfg	3:0	Alpha rb cold -- Filter aggressiveness control for Red/Blue Cold pixels.
	7:4	Alpha rb hot – Filter aggressiveness control for Red/Blue Hot pixels.
	11:8	Alpha g cold – Filter aggressiveness control for Green Cold pixels.
		Alpha g hot – Filter aggressiveness control for Green Hot pixels.
	15:12	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
	31:16	the noise level (variance) to prevent noise from interfering with bad pixel detection.
cfg	0	Raw Format 0 = Planar mode, 1 = Bayer mode
	2:1	Bayer Order: 00: GRBG 01: RGGB 10: GBRG 11: BGGR
	3	Gr/Gb imbalance correction enable
	4	Hot/Cold pixel suppression enable
	5	Hot/Cold pixel suppression on Green pixels only
	6	AWB Patch statistics output enable
	7	Histogram statistics output enable
	11:8	Input data bit width minus one
	23:16	Gr/Gb bad pixel detect threshold. The Gr/Gb imbalance filter includes a relatively crude bad pixel detection mechanism. This is to prevent Gr/Gb imbalance from mixing hot or cold pixels with "good" pixels. This bad pixel detection logic detects if a pixel is an

Name	Bits	Description
		outlier with respect to the other pixels in the neighbourhood. The programmable threshold is used when determining if a pixel is an outlier. The detection logic operates on Green pixels only, since it is part of the Gr/Gb imbalance filter. Gr/Gb correction will be bypassed if any hot or cold pixels are detected in the 5x5 neighbourhood. This feature is independent from the Hot/Cold suppression filter, which is more sophisticated.
gainSat[2][4]	 15:0 31:16	4 gains for each 2 lines Gain value applied to pixels. Format is U8.8. Level at which pixels will be clamped after corresponding gain is applied. Format is U16.
oBufStats	UInt32 *	Private
statsBase	UInt32 *	Base address of statistics buffers; 8byte (64bit) aligned.
statsPatchCfg	 5:0 13:8 23:16 31:24	Accumulation patch configuration X_No – number of patches in horizontally - 1 Y_No – number of patches in vertically – 1 Patch width – 1 (programmed as patch width minus 1) Patch height – 1 (programmed as patch height minus 1)
statsPatchStart	15:0 31:16	X coordinate Y coordinate
statsPatchSkip	15:0 31:16	X skip – 1 (programmed as patch width minus 1) Y skip– 1 (programmed as patch height minus 1)
statsFrmDim	15:0	Frame Width; Only 1 line is ever output per frame
statsPlanes	 3:0 7:4 11:8 15:12 19:16 21:20	For non-Bayer data 4 Patch Planes may be programmed. For-Bayer data only the first patch plane will be active. Patch Plane 0 Patch Plane 1 Patch Plane 2 Patch Plane 3 Histogram Plane Active patch planes - 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 = 01 and program the required planes into Patch Plane 0 and Patch Plane 1.

7.7 Demosaic / Debayer filter

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_DBYR_ID	U8, U16	U8, U16	DbyrParam	11

Input	Single plane of Bayer data in up to 16 bits
Operation	Local luma-adaptive hybrid Bayer de-mosaicing filter combining an optimized implementation pf AHD and bilinear interpolation
Input buffer	Minimum of 11 lines
Output	Planar RGB in up to 16 bits
Instances	1

This filter converts raw Bayer data into 3-channels-per-pixel RGB data. The input and output data formats can be separately configured to be between 6 and 16 bits. When a width of 8 bits or less is specified, the corresponding input our output buffer must be 8 bit, and each pixel of data shall be packed into the LSB of a single byte. When a data width of 9 or more is specified, the corresponding input or output buffer must be 16 bit, and each pixel of data shall be packed into the LSB of a byte pair.

The filter can be configured to demosaic the green channel only, ignoring pixels at Red and Blue locations. This is useful for processing data from certain non-Bayer CFAs. To enable this mode, set the “Luma Only” and “Force RB to zero” bits.

NOTE: Unlike most other filters, the demosaic filter does not perform automatic pixel replication at image borders. Since the filter has a kernel size of 11x11, the input image should be pre-padded with a 5-pixel border. The output image will be 10 pixels smaller that the input image in each dimension.

7.7.1 Features

7.7.1.1 De-worming

“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 neighbourhood 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.

7.7.2 DbyrParam Configuration

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

Name	Bits	Description
frmDim		PRIVATE - frame dimensions in pixels
	15:0	Frame width
	31:16	Frame height
cfg	1:0	Bayer Order: (for RGBW data set this register to 00 - GRBG) 00: GRBG 01: RGGB 10: GBRG 11: BGGR
	2	Luma only. Set this bit when demosaicing the green channel only.
	3	Force RB to zero. Set this bit when demosaicing the green channel only.
	7:4	Input data bit width minus one
	11:8	Output data bit width minus one
	14:12	Output plane channel order: 000 – RGB 001 – BGR 010 – RBG 011 – BRG 100 – GRB 101 – GBR
	16:15	Plane multiple **pm** for processing multiple planes of Bayer data. Number of output planes minus 1 (set to two for normal Bayer demosaicing).
	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. Used for de-worming (*gradient_mult* in equation 1 above). Value is 1.7 Fixed Point.
thresh		These thresholds are used in color Moire avoidance
	12:0	abs_th1
	24:13	abs_th2
dewormCfg	15:0	De-worming offset (*offset* in equation 1 above). In unsigned 8.8 fixed point format.
	31:16	De-worming slope (*slope* in equation 1 above). 16-bit signed integer.

7.8 Sharpen / 7x7 separable convolution

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_SHARPEN_ID	U8, FP16	U8, FP16	SharpParam	7

Input	Up to 16 planes (sequentially) of FP16/U8F
Operation	Enhanced unsharp mask. Programmable (separable, symmetric) blur filter kernel. Sharpening functionality can be disabled to use filter kernel on its own.
Input buffer	Minimum number of lines in buffer must matched programmed dimensions of filter kernel (3x3, 5x5 or 7x7)
Output	Sharpened image, filtered image or delta (mask) in up to 16 planes of FP16/U8F
Instances	1

The Sharpen filter enhances image sharpness, which avoiding many of the problems typically associated with sharpening, including overshoot/undershoot, halos and noise amplification. It contains limits to constrain the sharpened pixel to be within a programmable percentage of the minimum and maximum values in the pixel neighborhood. Furthermore, to prevent giving the image an unnatural look, the constrained sharpened value may be mixed with the unconstrained sharpened value, using a programmable mix factor. Sharpening may be restricted to the midtones. The location of the midtone range is configured via a set of programmable stops. Finally, a noise limiter prevents noise in largely uniform areas from being amplified. This limiter is controlled via a programmable threshold.

The sharpen filter can also be put into a convolution-only mode, whereby it can be used to apply a convolution kernel with programmable coefficients (provided the filter is separable, and that the horizontal filter kernel is the same as the vertical filter kernel). In normal sharpening mode, the filter is programmed with coefficients which define a Gaussian kernel, with a kernel size of up to 7x7. The filter operation is equivalent to a 1D vertical filter followed by a 1D horizontal filter. The mapping of the user-specified coefficients to kernel coefficients is shown in the tables below.

C0	C1	C2	C3	C2	C1	C0

Figure 18: Sharpen Filter horizontal kernel

C0
C1
C2
C3
C2
C1
C0

Figure 19: Sharpen Filter vertical kernel

7.8.1 Features

7.8.1.1 Limiting sharpening to the mid-tones

Sharpening can be restricted to the midtones via a set of 4 programmable stops. The areas outside of the first and last stop will not be sharpened at all, whereas full sharpening is applied to the range specified by the middle two stops. The figure below shows how the sharpening strength is attenuated as a function of local brightness, under the control of the programmable stops (the stops in this example are set to .05, .15, . 75 and 1).

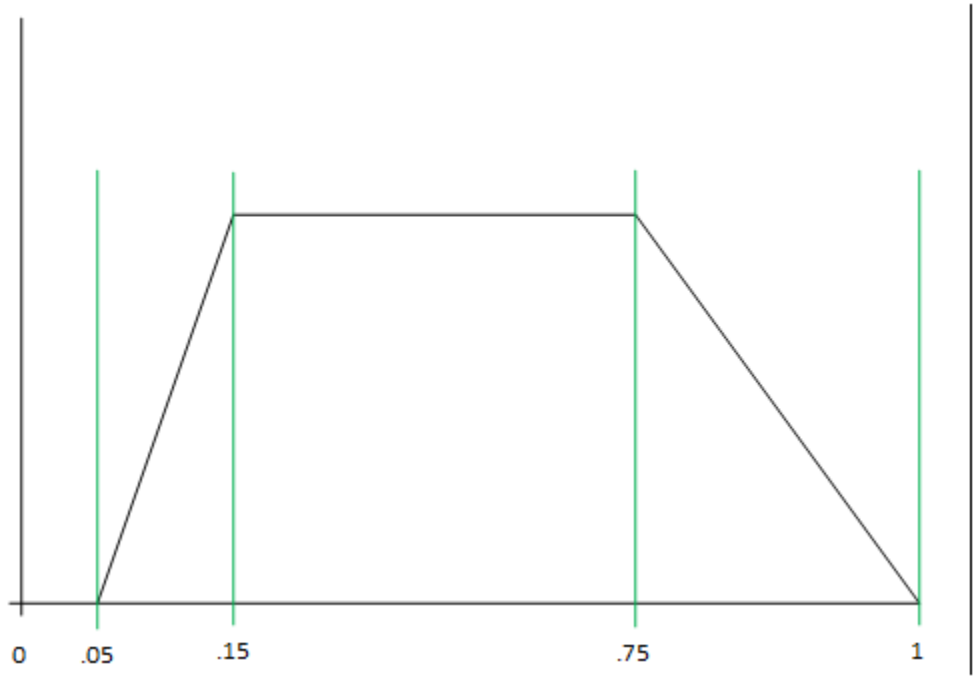

Figure 20 Programmable stops to limit sharpening to the mid-tones

7.8.2 SharpParam Configuration

The Sharpen filter is configured via the **Sharp*Param*** structure, which has the following user-specifiable

fields:

Name	Bits	Description
frmDim		PRIVATE - frame dimensions in pixels
	15:0	Frame width
	31:16	Frame height
cfg	2:0	Kernel size, **k**, in range (3 <= **k** <=7).
	3	Clamp output. Set to 1 to clamp the filter FP16 output into the range [0, 1.0]
	4	Mode 0 = sharpening. 1 = convolution mode (bypass sharpening and use as a generic, symmetric, 7x7 filter kernel (e.g. for blur)
	5	Output Deltas only; 0 = output sharpened pixels (input + delta) 1 = output only the delta between input pixels and sharpened pixels (valid only when the output format is FP16).
	31:16	Threshold **t** (FP16)**.** Minimum threshold. If the delta between the sharpened and the non-sharpened data is below this threshold, no sharpening is applied.
strength	15:0	Sharpen Strength Positive - when the pixel is to be made brighter, in FP16 format. Range is [0, 1.0].
	31:16	Sharpen Strength Negative - when the pixel is to be made darker, in FP16 format. Range is [0, 1.0].
clip	15:0	Sharpening Clipping Alpha, **alpha** (FP16). Mix factor for mixing constrained sharpened pixel with unconstrained sharpened pixel. Format is FP16, and range is [0, 1.0]. A value of 1 takes 100% of the constrained value, and a value of 0 takes 100% of the unconstrained value.
limit	15:0	Undershoot limit control. The sharpened value is constrained to being no less than MIN*N, where MIN is the smallest (input) pixel value in the 3x3 neighbourhood, and N is the value programmed into this field. The range of N is [0, 1.0]. Format is FP16.
	31:16	Overshoot limit control. The sharpened value is constrained to being no greater than MAX*N, where MAX is the largest (input) pixel value in the 3x3 neighbourhood, and N is the value programmed into this field. The range of N is [1.0, 2.0]. Format is FP16.
rngStop01	15:0	Range Stop value 0 (FP16).
	31:16	Range Stop value 1 (FP16).
rngStop23	15:0	Range Stop value 2 (FP16).
	31:16	Range Stop value 3 (FP16). Range stop values should increase monotonically, and be in the range [0, 1.0].
coeff01	15:0	Filter Coefficient 0 (FP16)
	31:16	Filter Coefficient 1 (FP16)
coeff23	15:0	Filter Coefficient 2 (FP16)
	31:16	Filter Coefficient 3 (FP16)

7.9 Luma Denoise

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_LUMA_ID	U8, FP16	U8F, FP16 (Luma) U8 (Reference)	YDnsParam	7 (Input) 11 (Ref)

Input	Up to 16 planes (sequentially) of FP16/U8F *input* & U8 *reference* input
Operation	Luma denoise using an optimized implementation of the non-local means algorithm
Input buffers	Minimum of 7 lines of *input* and 11 lines of *reference*
Output	Up to 16 planes of FP16/U8
Instances	1

This filter is designed to remove noise from a single image plane. It is based on the non-local means algorithm. It uses a 7x7 search area, and a 5x5 similarity window. The filter takes two inputs: a reference image, in addition to the image to be denoised. This gives a lot of flexibility with respect to how the filter can be deployed. For example, excellent results can be obtained if denoise is applied to an image which has already been sharpened, whereas sharpening has not been applied to the reference image. In fact, it can be an advantage if the reference image is slightly blurred beforehand.

Furthermore, the reference image can be modified to give the user control over the denoise strength. For example, by applying a gamma-like transform to the reference image, the user can control how strongly the denoise is applied to different parts of the pixel intensity range. Also, spatial variation is possible: for example, the reference image could be attenuated closer to the corner of the image, meaning that the corners will be denoised more aggressively (which is often desired, since lens shading compensation will amplify the edges of the image, where less light has been collected, making them look more noisy).

The filter's output at a given pixel location is a weighted average of the pixels in the 7x7 neighborhood around the input pixel. The weights are calculated based on the reference input. For each position in the 7x7 neighborhood, a similarity is calculated between the 5x5 area around that position compared to the center pixel's position. A function (defined via a programmable LUT, and usually a Gaussian) is then applied to the similarity metric, to get the weight at that position. Finally, the weight may be modified by a shift operation, such that pixels closer to the center pixel are assigned larger weights.

This filter requires 11 lines of U8 reference data, and 7 lines of U8 or FP16 input data.

7.9.1 YDnsParam Configuration

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

Name	Bits	Description
frmDim	 15:0 31:16	PRIVATE - frame dimensions in pixels Frame width Frame height
cfg	3:0	Bit position which, in addition to the 32-entry LUT, defines the weight adjustment function.

Name	Bits	Description
	15:8	Mix factor which blends the input back into the denoised output. 0 = output is the same as input, 255 = do not mix any of the input back into the output.
gaussLut[4]	31:0	Pointer to a 32-entry LUT, which is an array of U8's. Each value must be <= 15.
f2	1:0	Shift value at position 0,0. The weight at the corresponding locations is shifted left by the specified number of bits (0, 1, 2 or 3).
	3:2	Shift value at position 0,1.
	5:4	Shift value at position 0,2.
	7:6	Shift value at position 0,3.
	9:8	Shift value at position 1,0.
	11:10	Shift value at position 1,1.
	13:12	Shift value at position 1,2.
	15:14	Shift value at position 1,3.
	17:16	Shift value at position 2,0.
	19:18	Shift value at position 2,1.
	21:20	Shift value at position 2,2.
	23:22	Shift value at position 2,3.
	25:24	Shift value at position 3,0.
	27:26	Shift value at position 3,1.
	29:28	Shift value at position 3,2.
	31:30	Shift value at position 3,3.

7.10 Chroma Denoise

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_CHROMA_ID	U8	U8	ChrDnsParam	11

Input	Up to 16 planes of U8 chroma difference data sequentially or 3 planes in parallel
Operation	Chroma denoise using wide cascaded, thresholded box filters. Filtering is performed first vertically then horizontally Optional *reference* input in sequential processing configuration
Input buffer	Minimum of 21 lines
Output	Planar chroma difference data
Instances	1

The Chroma filter is a weighted averaging filter, and operates on U8 data. It requires 21 lines of input data. It typically operates on chrominance data that has been subsampled both horizontally and vertically by half. In the horizontal direction, pixels are processed by up to three separate filters that run in series. The widths of these filters are 23, 17 and 13. They may each be independently enabled or disabled. If all three horizontal filters are enabled, the effective horizontal kernel size is 53. Typically, only a single filter is enabled in good lighting conditions, and more filters are enabled in lower light conditions. The filter may optionally be configured to take a reference image that is separate from the image being denoised. The filter performs a weighted average of pixels in the neighborhood, and the reference image (or the main input image if no

reference image is provided) is used to compute the weights.

The filter can operate in single-plane mode, or three-plane mode.

7.10.1 Features

7.10.1.1 Single-plane mode

In single-plane mode, a single plane of 8-bit data is read and processed. Filtering is controlled by two programmable thresholds for each direction (horizontal and vertical): “t1” and “t2”. When computing weights, the center pixel is compared against each pixel in the neighborhood. If the absolute difference between the center pixel and the neighborhood pixel is less than threshold “t1”, that neighborhood location receives a weight score of 1. If it’s also bigger than “t2”, then the location receives a weight score of 2. “t2” should be set >= “t1”.

7.10.1.2 Three-plane mode

In three-plane mode, three planes of 8-bit data are read and processed simultaneously. Filtering is controlled by three programmable thresholds for each direction (horizontal and vertical): “t1”, “t2” and “t3”. These weights are used when calculating weights for the first, second, and third planes, respectively. For each plane, the center pixel is compared against each pixel in the neighborhood. The neighborhood location receives a weight score of 1, only of the absolute difference is below the corresponding threshold, for all three planes. Otherwise, the neighborhood pixel receives a weight score of 0.

7.10.1.3 Difference limit

There is also a programmable limit, which limits the difference between the filter input and the filter output at a given pixel location.

7.10.2 ChrDnsParam Configuration

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

Name	Bits	Description
frmDim		PRIVATE - frame dimensions in pixels
	15:0	Frame width. Must be >=**23**
	31:16	Frame height. Must be >=**11**
cfg	2:0	Bit 0 corresponds to first horizontal pass, bit 1 corresponds to second and bit 2 to third pass.
	3	Enable separate reference input. If Reference Enabled, then must be in Single Plane Mode. 0 – Reference not enabled 1 – Reference Enabled
	11:4	Limit - the difference between the input value and the output value at any given pixel location is limited to this amount.
	12	Force all weights in horizontal direction
	13	Force all weight in vertical direction
	14	Enable three-plane mode. If three-plane mode is enabled, bit 3 (Enable separate reference input) must be 0.
thr[2]		Thresholds t1 and t2 are used differently for Single Plane Mode and Three Plane Mode. In Single Plane Mode two thresholds are used on the active plane. In Three Plane mode one threshold is defined per plane. In Single Plane Mode, at least one of horizontal (t1 or t2) and one of the vertical (t1 or t2) must be

Name	Bits	Description
		non zero. In Three Plane Mode both all thresholds must be non zero.
		thr[0]
		Horizontal threshold **t1**
	7:0	Horizontal threshold **t2**
	15:8	Vertical threshold **t1**
	23:16	Vertical threshold **t2**
	31:24	**thr[1]**
		Horizontal threshold **t3**
	7:0	Horizontal threshold **t3**
	23:16	

7.11 Median

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_MED_ID	U8	U8	MedParam	3,5,7

Input	Up to 16 planes (sequentially) of U8 data
Operation	Classic 2D median filter with configurable kernel size, 3x3, 5x5 or 7x7
Input buffer	Minimum number of lines in buffer must match programmed dimensions of filter kernel (3x3, 5x5 or 7x7)
Output	Up to 16 planes (sequentially) of U8 data. 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 operates on U8 data, and can be configured with a kernel size of 3x3, 5x5 or 7x7. It can also be configured to perform some other operations such as erode and dilate. Internally, the filter maintains a sorted list of all pixels in the **k**x**k** neighborhood, where **k** is the kernel size. An output select parameter determines which entry in the sorted list is used as the output. To perform median filtering, set this to floor(**k**x**k**/2). To perform erode, set it to zero, and to perform dilate, set it to **k**x**k**-1.

The median filter implementation uses a progressively updated median table approach. The filter consumes columns of pixels and maintains an ordered table of pixels, sorted by value. The list order is updated progressively as new columns are read in and old columns are evicted.

The filter implementation effectively sorts the input pixel kernel – the central pixel in the list is the median value of the kernel. It is also possible to output any other value in the list via a programmable output selection register (default selection is the central/median value). This means the filter may be configured to implement erode/dilate as follows.

Erode: The value of the output pixel is the minimum value of all the pixels in the input pixel's neighbourhood, i.e. set selection to zero.

Dilate: the value of the output pixel is 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 (e.g. 7x7) minus one.

A programmable threshold is also provided allowing selective filtering of pixel values; only pixels greater than the threshold value are filtered. Pixels 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.

7.11.1 MedParam Configuration

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

Name	Bits	Description
cfg	2:0 13:8	Kernel size, **k**. Must be 3, 5 or 7. Output selection. Value range is [0, kxk-1]. Set to floor(kxk/2) to perform median filtering. 0 to perform the erode operation, and to

Name	Bits	Description
	24:16	kxk-1 to perform dilate. 9-bit signed threshold, **t**. Pixels below this threshold will not be modified. Specify -1 (0x1ff) to disable the threshold feature (all pixels are filtered), this is the default value. Setting to 255 puts the filter into bypass mode.

7.12 Polyphase FIR Scaler

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_UPFIRDN_ID	U8, FP16	U8, FP16	PolyFirParam	7

Input	Up to 16 planes (sequentially) of FP16 or U8F
Operation	Up/downscale with separable 3, 5, 7-tap, 16-phase FIR filter
Input buffer	Minimum number of lines in buffer must match programmed dimensions of filter kernel (3x3, 5x5 or 7x7)
Output	Up to 16 planes of FP16 or U8F
Instances	1

The poly-phase FIR filter scaler is suitable for high-quality implementations of scaling using e.g. Lanczos re-sampling.

Image scaling in a given direction (horizontal or vertical) is specified by a factor which is a ratio N/D (Numerator/Denominator), where N is a 5 bit integer in the range [1,16] and D is a 6 bit integer in the range [1,63]. The SIPP poly-phase FIR filter supports effective filter kernels of up to 112-taps organized as 16 phases of 7-taps.

7.12.1 Polyphase FIR Features

The Polyphase filter is performed in three steps:

- Upsample the input by a factor of N using zero-insertion
- Interpolate the upsampled input data by low-pass filtering
- Downsample the result by a factor of D

This process is often known as Up/FIR/Down, or upfirdn.

An interpolating poly-phase FIR filter may implement very high order FIR filters efficiently by taking advantage of the fact that after upsampling many filter inputs are zero and therefore do not contribute to the filter output. The SIPP poly-phase FIR filter supports effective filter kernels of up to 112-taps organized as 16 phases of 7-taps.

7.12.1.1 Upsampling and filtering

Figure 21 shows horizontal upsampling by a factor of 2. The input pixels are p0, p1, p2, p3, p4, p5, p6, p7 and so on. There will be twice the number of output pixels, e.g. p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13 and so on. The diagram shows the input pixel stream after zero-insertion. The (zero-padded) input pixel locations filtered to produce the first 3 output pixels are shown.

The filter used has 14 effective taps but notice how, for the first output pixel p0, all odd filter inputs are zero and will therefore not contribute to the result. For the second output pixel p1 all even filter inputs are zero.

The 14-tap filter function may therefore be decomposed into two 7-tap filters. The filter inputs are held for 2 filter *phases* and the filter coefficients used are switched on each *phase*: the 7 even filter coefficients are used in phase 0; the 7 odd filter coefficients are used in phase 1.

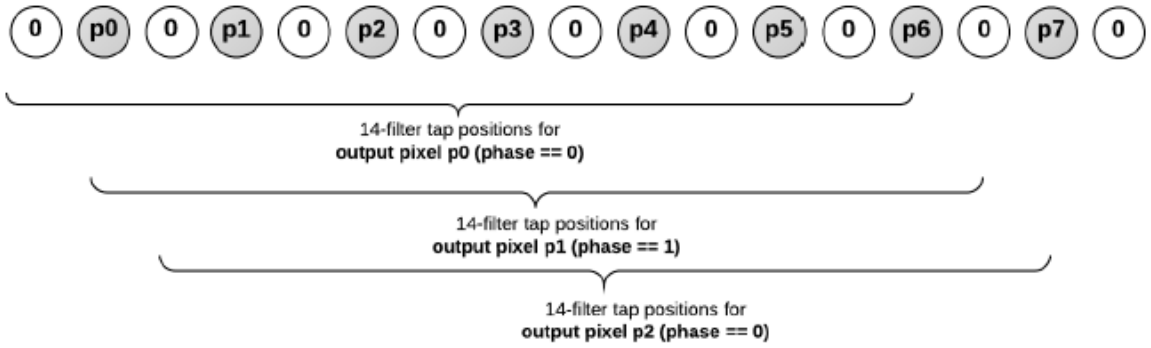

Figure 21: Upsampling by a factor of 2 and filtering with 2 phases

E.g. if the filter coefficients are h0, h1, h2… h12, h13:

- Output pixel p0, phase == 0: filter p0, p1 … p5, p6 using even coefficients h0, h2… h10, h12
- Output pixel p1, phase == 1: filter p0, p1 … p5, p6 using odd coefficients h1, h3… h11, h13
- Output pixel p2, phase == 0: filter p1, p2 … p6, p7 using even coefficients h0, h2… h10, h12

Figure 22 shows horizontal upsampling by a factor of 3. The input pixels are p0, p1, p2, p3, p4, p5, p6, p7 and so on. There will be three times the number of output pixels, e.g. p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14, p15, p16, p17, p18, p19, p20 and so on. The diagram shows the input pixel stream after zero-insertion. The (zero-padded) input pixel locations filtered to produce the first 4 output pixels are shown.

The filter used has 21 effective taps, implemented using a 7-tap filter, holding the filter inputs for 3 phases and switching the coefficients which are used on each phase.

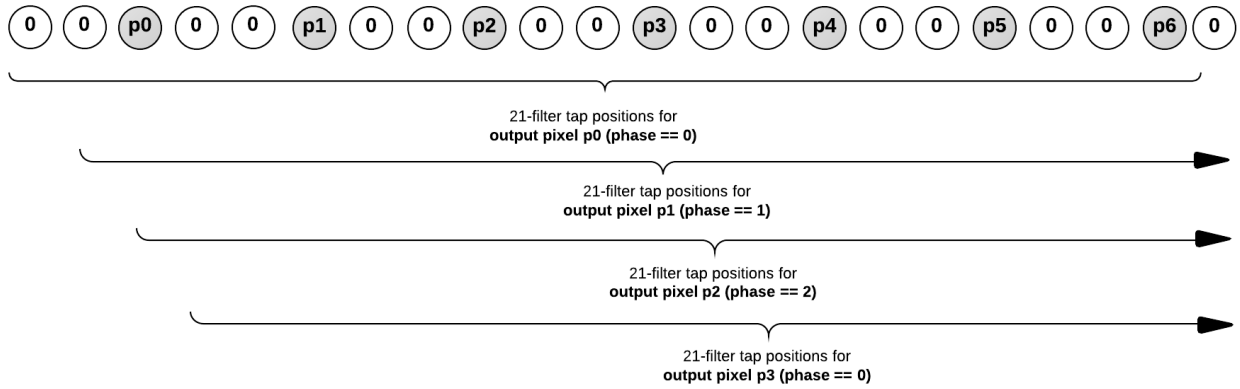

Figure 22: Upsampling by a factor of 3 and filtering with 3 phases

E.g. if the filter coefficients are h0, h1, h2… h19, h20:

Output pixel p0, phase == 0: filter p0, p1 … p5, p6 using coefficients h0, h3… h15, h18

Output pixel p1, phase == 1: filter p0, p1 … p5, p6 using coefficients h1, h5… h17, h20

Output pixel p2, phase == 2: filter p0, p1 … p5, p6 using coefficients h2, h4… h16, h19

Output pixel p3, phase == 0: filter p1, p2 … p6, p7 using coefficients h0, h3… h15, h18

The above examples show horizontal upsampling but the procedure is identical in the vertical direction. Note also that the examples shown should be considered to represent the pixel locations filtered at an

arbitrary point in an image scanline and that at the left/right edges image is padded by replication of the first/last pixel on the line, respectively, (prior to zero-insertion).

7.12.1.2 Downsampling

After upsampling and filtering downsampling is accomplished by decimation by the programmed rate (the denominator D of the scale factor in the range [1,63]).

7.12.1.3 Filter construction

The SIPP 2D poly-phase FIR filter implementation is constructed by cascading two 7-tap 1D FIR filters, as shown in Figure 23. The number of phases in use corresponds directly to the upsampling factor (N). The maximum upsampling factor is 16 (effectively allowing a 112-tap filter). Input lines are scaled vertically first (after padding by replication at the top/bottom) followed by horizontally (after padding by replication at the left/right).

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 downscaled output are actually read and filtered. The pixels of the current output line from vertical scaling are then horizontally scaled and filtered. 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 downscaled output are filtered before being written to the output buffer (i.e. some filter phases are skipped based on the programmed downscaling factor).

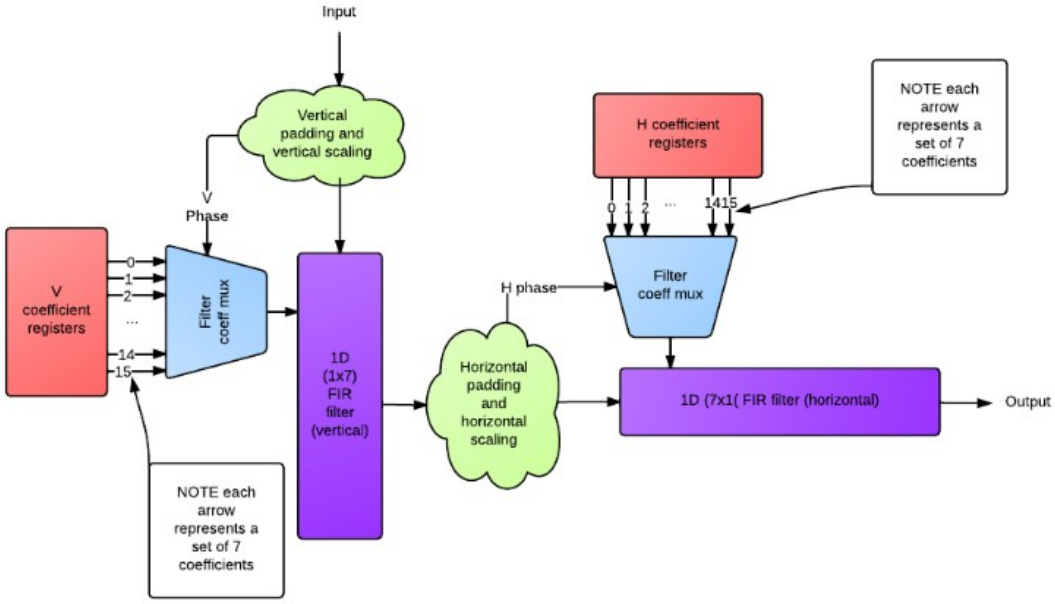

Figure 23: Polyphase FIR construction

7.12.2 Polyphase FIR Configuration

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 upscaling 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].

For example, if we wish to downscale 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 we are effectively specifying that we will use 12 of the 16 available phases to low pass filter our upscaled (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 either direction is configurable to 3, 5 or 7. 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.

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

Name	Bits	Description
frmDimPar	15:0 31:16	Input frame width in pixels (must be >=**k**) Input frame height in pixels(must be >=**k**)
frmDimFlt	15:0 31:16	Output frame width in pixels (must be >=**k**) Output frame height in pixels(must be >=**k**)
mode	UInt32	0 = Auto; 1 = Advance
autoType	UInt32	0 = Lanczos; 1 = Bicubic; 2 = Bilinear
clamp	UInt32	Set to 1 to clamp the filter FP16 output into the range [0, 1.0]
horzD	UInt32	Horizontal scale factor denominator. Maximum valid value is 63.
horzN	UInt32	Horizontal scale factor numerator. Maximum valid value is 16
vertD	UInt32	Vertical scale factor denominator.

Name	Bits	Description
vertN	UInt32	Vertical scale factor numerator. Maximum valid value is 16
horzCoefs	Uint16*	Pointer to list of Horizontal coefficients
vertCoefs	Uint16*	Pointer to list of Vertical coefficients

7.13 LUT

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_LUT_ID	U8, U16, FP16	U8, U16, FP16	LutParam	1

The LUT is a highly flexibly lookup table which can be used for tonal curve or gamma application, or other complex functions. In FP16 mode, interpolation is used to minimize quantization or precision loss.

In FP16 mode, the input must be normalized to the range [0, 1.0]. The exponent of the input is used to select from 16 ranges. The amount of precision in each range is independently configurable, allowing the needed precision to be obtained with optimal use of the LUT's dedicated memory.

In planar mode, the LUT can be configured to either apply the same LUT to all three planes or to apply a separate LUT to each plane.

7.13.1 LUT Features

The LUT filter has two main operation modes Normal mode and Channel Mode.

In **Normal mode** the read client operates on 1 line of data at a time. The read client populates the pixels from the 1 read line into 1 of the Index generation blocks (Index Generation block 2) one pixel at a time. This Index generation block generates 1 index into the LUT RAM and reads the corresponding value(s) from the LUT ram and passes them to the interpolation block. If interpolation is required 2 values are read from the RAM simultaneously and interpolation between the two values is performed in the Interpolation block. If Interpolation is not required then 1 value is read from the RAM and the value is passed through the interpolation block (Interpolation block 2). The resulting pixel data is then passed to the write client for output. Multiple Planes of data can be processed in the normal fashion whereby the LBRC reads from different planes on a line by line basis and the write client writes to different planes on a line by line basis (I.e. a full line of Plane 0 is processed followed by a full line of Plane 1 and so on)

In **Channel Mode** the read client is set up to read from up to 4 different planes at a time. These n different planes will be output from the read client on n different lines simultaneously and the pixel data for the n lanes will be passed on simultaneously to the n index Generation blocks each simultaneously generating an index into the LUT RAM. The n indexes will read 2n LUT values if interpolation is required and n LUT values if interpolation is not required. The values will be passed on to the n interpolation blocks and processed before being loaded into the planar write client. This planar write client will write the data from the n channels to n different planes by interleaving the accesses to the AMC. (64 bytes from channel 0 followed by 64 from channel 1 followed by 64 from channel2 and so on). The Read Client can be configured to read from up to 4 different planes on a line by line basis also. The total number of planes can't exceed 16.

7.13.2 LUT Configuration

Name	Bits	Description
frmDim		**PRIVATE** - Frame dimensions in pixels
	15:0	frame width
	31:16	frame height
cfg	0	Interpolation mode 0 – Integer mode 1 – FP16 mode
	1	Channel Mode – set to 1 to enable
	7:3	Integer width (if Integer mode) – 8 to 16
	11:8	Number of LUTs per channel
	13:12	Number of active channels
	14	Enable LUT load
	15	APB access enable
sizeA		LUTSizes7_0
	31:28	LUT region 7 size index
	27:24	LUT region 6 size index
	23:20	LUT region 5 size index
	19:16	LUT region 4 size index
	15:12	LUT region 3 size index
	11:8	LUT region 2 size index
	7:4	LUT region 1 size index
	3:0	LUT region 0 size index
sizeB		LUTSizes15_8
	31:28	LUT region 15 size index
	27:24	LUT region 14 size index
	23:20	LUT region 13 size index
	19:16	LUT region 12 size index
	15:12	LUT region 11 size index
	11:8	LUT region 10 size index
	7:4	LUT region 9 size index
	3:0	LUT region 8 size index
lut	void*	Pointer to U8 or FP16 look-up-table

7.13.3 LUT Usage

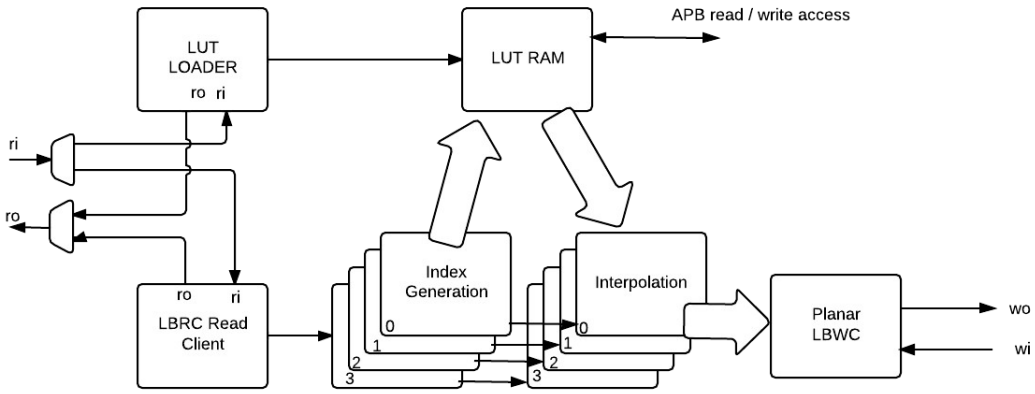

Figure 24: LUT Filter Structure

Figure 24 illustrates the underlying structure of the LUT Filter. The LUT Loader and the Line Buffer Read Client (LBRC) share a common Accelerated memory Controller (AMC) read port. This is because the LUT RAM will be loaded prior to any processing and hence the LUT Loader and the LBRC will never be active together. The Planar Line Buffer Write Client (LBWC) can write up to 4 planes of data at a time when the LUT is in channel mode.

7.13.4 LUT Modes

The LUT filter has two main operation modes Normal mode and Channel Mode.

In **Normal mode** the LBRC read client operates on 1 line of data at a time. The read client populates the pixels from the 1 read line into 1 of the Index generation blocks (Index Generation block 2) one pixel at a time. This Index generation block generates 1 index into the LUT RAM and reads the corresponding value(s) from the LUT RAM and passes them to the interpolation block. If interpolation is required 2 values are read from the RAM simultaneously and interpolation between the two values is performed in the Interpolation block. If Interpolation is not required then 1 value is read from the RAM and the value is passed through the interpolation block (Interpolation block 2). The resulting pixel data is then passed to the LBRC Write client for output. Multiple Planes of data can be processed in the normal fashion whereby the LBRC reads from different planes on a line by line basis and the write client writes to different planes on a line by line basis (I.e. a full line of Plane 0 is processed followed by a full line of Plane 1 and so on)

In **Channel Mode** the LBRC read client is set up to read from up to 4 different planes at a time. These n different planes will be output from the LBRC on n different lines simultaneously and the pixel data for the n lanes will be passed on simultaneously to the n index Generation blocks each simultaneously generating an index into the LUT RAM. The n indexes will read 2n LUT values if interpolation is required and n LUT values if interpolation is not required. The values will be passed on to the n interpolation blocks and processed before being loaded into the Planar LBWC write client. This planar write client will write the data from the n channels to n different planes by interleaving the accesses to the AMC. (64 bytes from channel 0 followed by 64 from channel 1 followed by 64 from channel2 and so on). The Read Client can be configured to read from up to 4 different planes on a line by line basis also. The total number of planes can't exceed 16.

7.13.4.1 Local RAM organization

There is 16k Bytes of local RAM storage to hold the various LUT configurations. The two modes put different

restriction on the ordering of data when programming the LUT. The LUT data stored in memory must be stored contiguously starting from a 64 bit aligned address. Each value will be store in a 16 bit location.

In Normal Mode the data will be stored starting at Lut value 0 for the LUT of table 0 followed by Lut value 1 of the LUT of table 0 and so on up until the last value of the table 0. This will then be immediately followed by the at Lut value 0 for the LUT of table 1 if Planar Lut Mode is selected and so on.

In Channel Mode the data will be stored starting at Lut value 0 for the LUT of Channel 0 followed by at LUT value 0 for the LUT of Channel 1 followed by at Lut value 0 for the LUT of Channel 2 followed by at LUT value 0 for the LUT of Channel 3 followed by Lut value 1 for the LUT of Channel 0 and so on. Only 1 LUT per channel is currently supported. If less than 4 channels are required then the corresponding data should be set to 0.

In the Local RAM storage block 8 1k x 16 bit RAMs are used. This is to allow reading of 8 Lut values for 4 channels of pixel data requiring interpolation.

7.13.4.2 Table Structure and Index generation

Let's define a Look up Folder (LUF) as an array of Look up Tables (LUT) which can contain a LUT for each plane of data in the data stream. This LUF and all of its associated LUTs will be stored contiguously in CMX memory starting with the value at position 0 for the LUT of Plane 0 (LUT0). Each value in the LUT is 16 bits. The address of Position 0 of LUT0 is defined by **lut** field.

Each LUT in the LUF will have the same size configuration as defined by **sizeA**[31:0] and **sizeB**[31:0]. These define the LUT as having 16 regions each having a configurable number of 16 bit entries equal to 2**n where n(x) is the programmable range size for region x.

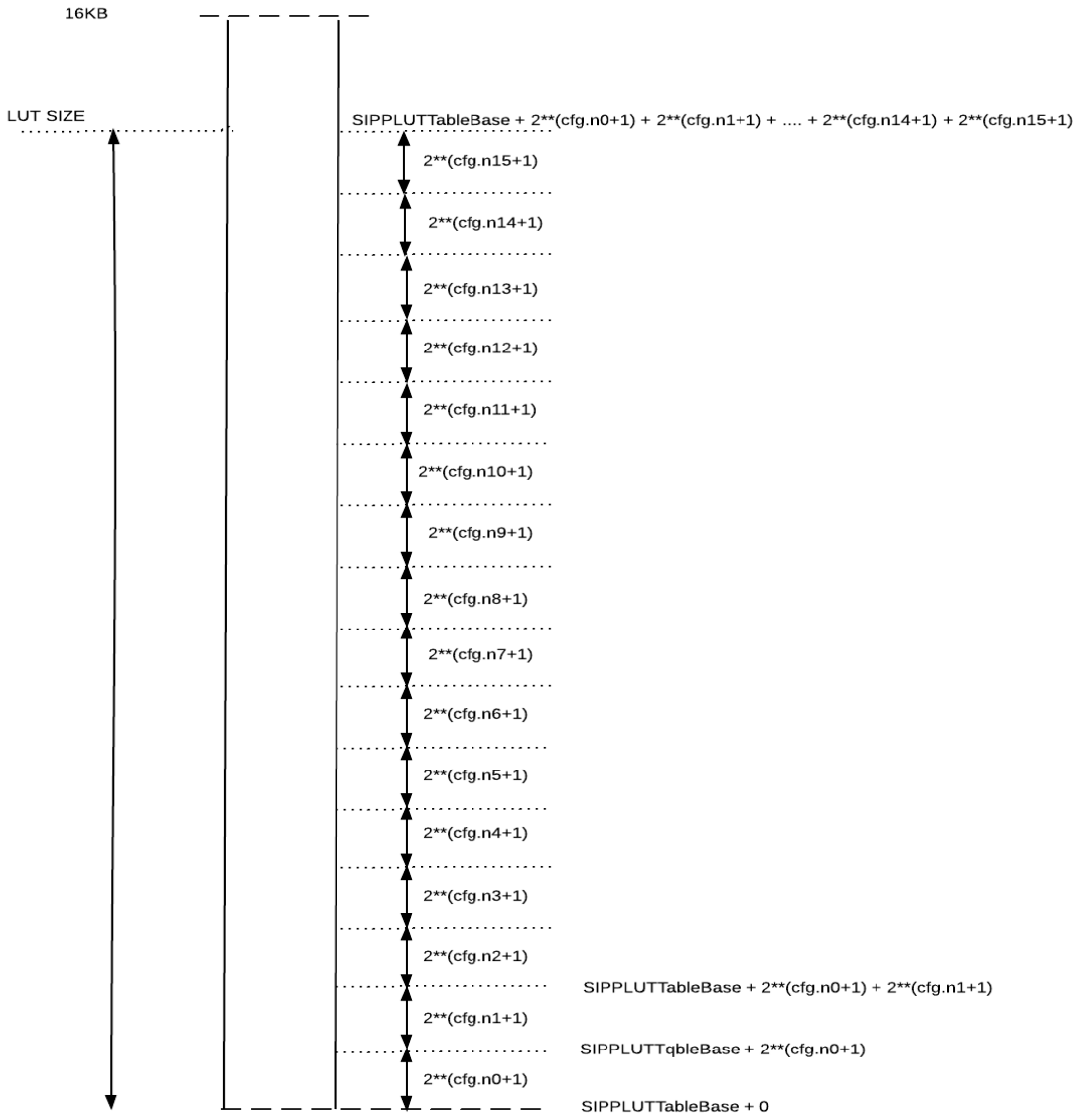

Figure 25: LUT Size Construction

The maximum LUT (or LUF) size is 16KB (8K 16 bit values). This limitation is set by the size of the local RAMs in hardware. The maximum number of entries in any range is 1024 and the minimum number of entries in any range is 1. Hence the values of n can range from 0 to 10. Each entry is 16 bits.

Range 0 of LUT0 starts at address **lut** and ends at location **lut** + (2**(n0+1) -2).

Range 1 starts at address (SIPPIBuf[19]Base [31:0] + 2**(n0+1)) and ends at address (**lut** + 2**(n0+1) + 2**(n1+1) - 2) as shown in Figure 26

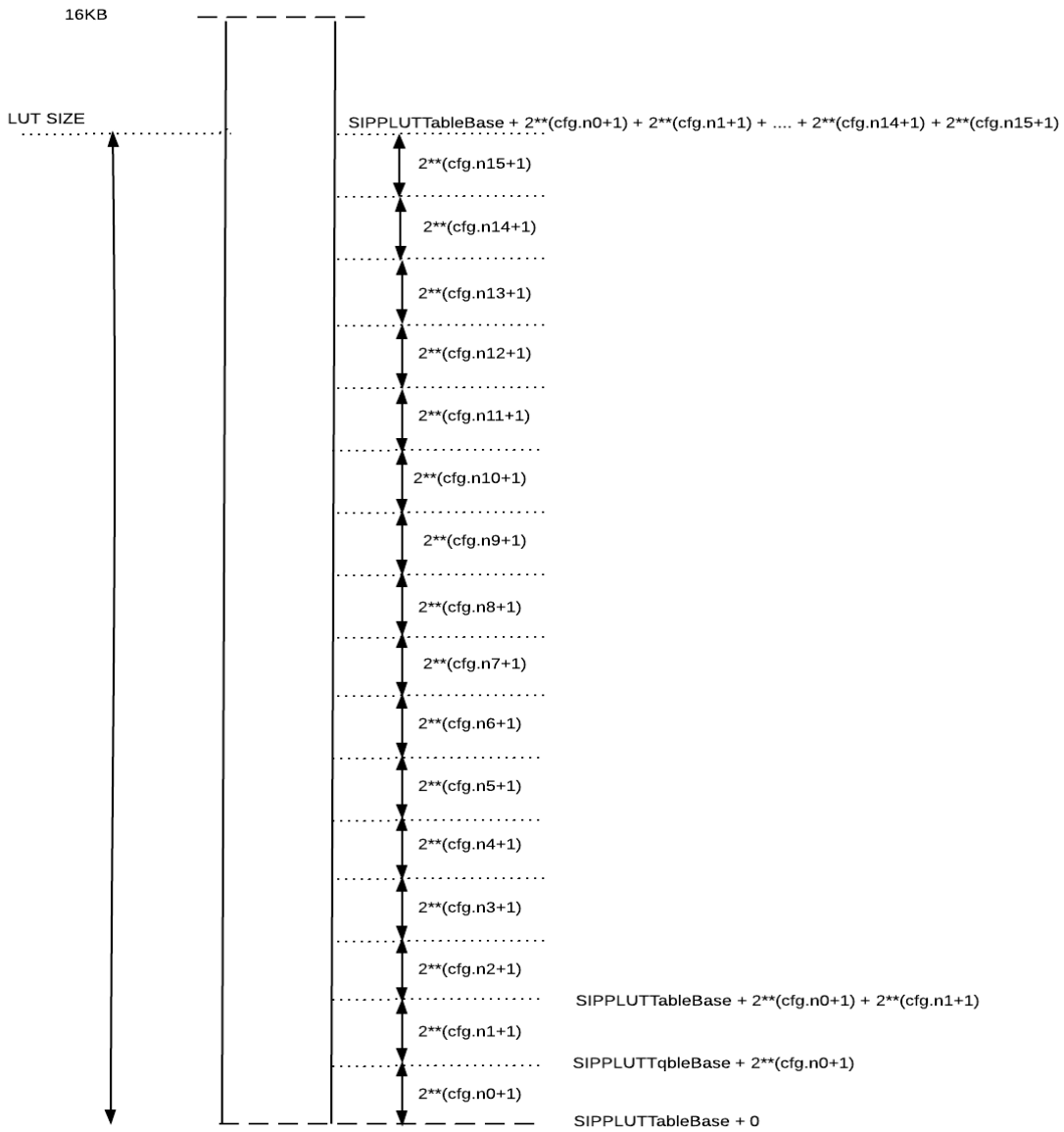

Figure 26: LUT Range

7.13.4.3 Multiple LUTs for planar data

More than 1 LUT can be stored in the local RAM if different LUTs are required per plane.

For multi-planar data which requires a different LUT per plane the entire LUF must fit within the 16KB limitation (For 16 Planes this will give 512 values per LUT).

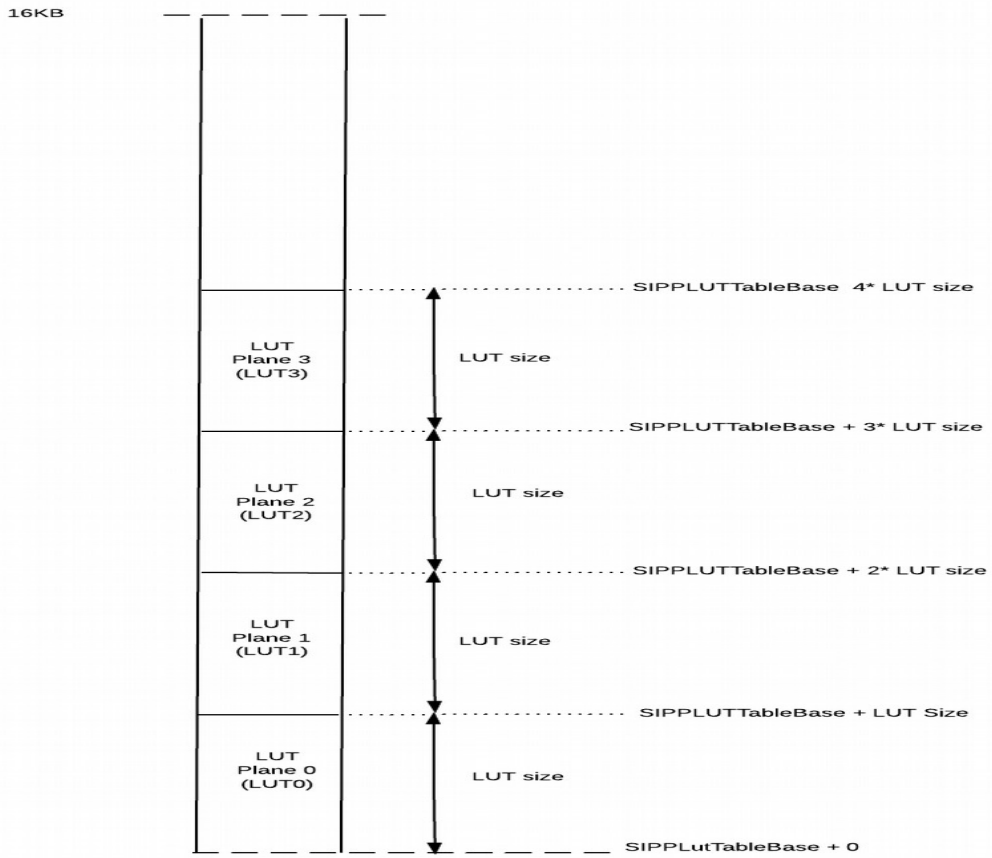

Figure 27: Multiple LUTs for Planar Data

7.13.4.4 LUT filter control, LUT address extraction and interpolation

The operation modes of this filter are controlled by **cfg**.

- **cfg**[0] selects between integer and FP16 formats. In integer mode **cfg**[7:3] represents the integer width. In FP16 mode linear interpolation between values will occur.
- **cfg**[1] controls whether or not a different LUT is used for each plane of multi-planar data. If this bit is set to 0 there will only be 1 LUT in the LUF and **cfg**[11:8] must be set to 4'b0000. **cfg**[1] is set to 1 then **cfg**[11:8] should be set to the maximum number of complete LUTs that are storeable in 8KB minus 1.
- **cfg**[14] is set for the first time that the **lut** field is set and tells the filter that the local RAM must be updated when a frame start is received. Subsequent changes to the **lut** will take effect following reception of the FrameEnd signal. This bit should be explicitly set to 0 before being set to 1.

7.13.4.5 Address extraction

The range selector for the table is dependent on the mode of operation of the table. In FP16 mode it is assumed that the incoming FP16 Data is normalized and hence the exponent e16[14:10] (where FP16 = {z,e16[4:0],f16[9:0]} will always be in the range 0-15. Hence e16[4:0] is used as the range selector. In integer mode the top four bits of the integer will be used. Thus if the integer width is 11 then bits [10:7] are as the range selector. This is shown in the table:

Input Mode	Integer width	Range Selection	Offset_raw[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]

Figure 28: LUT range selection bits

The Address Offset within the range is now dependent on both the range size and the remaining unused bits (Offset_raw[9:0]) of the incoming data as shown in the table below. If interpolation is required the Interpolation Numerator (IN) is constructed from the remaining unused bits of the data, if any.

Range Size Indicator	Size	Address Offset	Interpolation_Numerator[9:0] IN
0	1	0	Offset_Raw[9:0]
1	2	Offset_Raw[9]	{Offset_Raw[8:0],1'b0}
2	4	Offset_Raw[9:8]	{Offset_Raw[7:0],2'b0}
3	8	Offset_Raw[9:7]	{Offset_Raw[6:0],3'b0}
4	16	Offset_Raw[9:6]	{Offset_Raw[5:0],4'b0}
5	32	Offset_Raw[9:5]	{Offset_Raw[4:0],5'b0}
6	64	Offset_Raw[9:4]	{Offset_Raw[3:0],6'b0}
7	128	Offset_Raw[9:3]	{Offset_Raw[2:0],7'b0}
8	256	Offset_Raw[9:2]	{Offset_Raw[1:0],8'b0}
9	512	Offset_Raw[9:1]	{Offset_Raw[0],9'b0}
10	1024	Offset_Raw[9:0]	10'b0

Figure 29: Address offset and interpolation fraction

The address of the current LUT is then calculated from the Start Address of the LUT + range offset + address offset.

```
LUT address = LUT start address + range offset + address offset
```

Linear interpolation between current value and next value

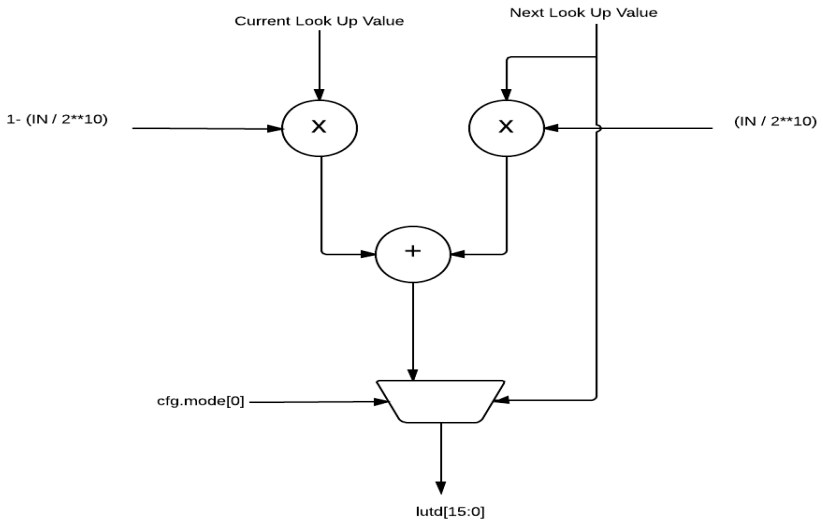

Figure 30: Linear interpolation

32 bits from this address are now read which will give the look up value for the current address and the current address + 1. This will allow for interpolation between the two values if required. Interpolation is only preformed on in FP16 mode. In Integer Mode the 16-bit current look up value is output from the filter.

7.14 Edge Operator

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_EDGE_OP_ID	U8	U8, U16	EdgeParam	3

Input	Up to 16 planes of U8/U16/FP16 image data or precomputed gradients
Operation	Flexible 3x3 edge-detection operator suitable for implementation of e.g. Sobel filter
Input buffer	Minimum of 3 lines
Output	Up to 16 planes of U8/U16/FP16 edge magnitude + angle
Instances	1

The Edge Operator filter is an extension of the standard Sobel filter functionality. The kernel convolution is first calculated to generate the horizontal and vertical derivative approximations as shown in

$$Gx = \begin{pmatrix} xcoef\_a & 0 & xcoef\_b \\ xcoef\_c & 0 & xcoef\_d \\ xcoef\_e & 0 & xcoef\_f \end{pmatrix} * A$$

$$Gy = \begin{pmatrix} ycoef\_a & ycoef\_b & ycoef\_c \\ 0 & 0 & 0 \\ ycoef\_b & ycoef\_e & ycoef\_f \end{pmatrix} * A$$

$$A = \begin{pmatrix} pixel[r-1][c-1] & pixel[r-1][c] & pixel[r-1][c+1] \\ pixel[r][c-1] & pixel[r][c] & pixel[r][c+1] \\ pixel[r+1][c-1] & pixel[r+1][c] & pixel[r+1][c+1] \end{pmatrix}$$

Figure 31: Edge operator convolution function

For Magnitude approximation, in order to calculate the square root function the following equations are used:

$$M = \sqrt{X^2 + Y^2} = |X| \sqrt{1 + \left({}^{Y}/_{X}\right)^2} \qquad eqn1$$

$$M = \sqrt{X^2 + Y^2} = |Y| \sqrt{1 + \left({}^{X}/_{Y}\right)^2} \qquad eqn2$$

Now if eqn1 is used when |X| > |Y| and eqn2 is used when |X| <= |Y| then an approximation for the function:

$$\left\{f(a) = \sqrt{1 + a^2} \qquad where\ 0 < a \leq 1\right\}$$

can be developed and used to approximate for M

The following approximation algorithm is used:

The range of a (i.e. 0 to 1) is divided into 16 equal sub ranges. f(a) is calculated for the top and bottom of each of theses ranges (17 values in total) using a square root approximation look-up table.

position	a	a^2	$1 + a^2$	$\sqrt{(1 + a^2)}$
1	0	0	1	1
2	0.0625	0.00390625	1.00390625	1.001951221
3	0.125	0.015625	1.015625	1.007782219
4	0.1875	0.03515625	1.03515625	1.017426287
5	0.25	0.0625	1.0625	1.030776406
6	0.3125	0.09765625	1.09765625	1.047690913
7	0.375	0.140625	1.140625	1.068000468
8	0.4375	0.19140625	1.19140625	1.091515575
9	0.5	0.25	1.25	1.118033989
10	0.5625	0.31640625	1.31640625	1.147347484
11	0.625	0.390625	1.390625	1.179247642
12	0.6875	0.47265625	1.47265625	1.21353049
13	0.75	0.5625	1.5625	1.25
14	0.8125	0.66015625	1.66015625	1.288470508
15	0.875	0.765625	1.765625	1.328768227
16	0.9375	0.87890625	1.87890625	1.370732012
17	1	1	2	1.414213562

The values for:

$$\sqrt{1 + a^2}$$

are stored in a look-up table in FP16 format. A linear interpolation between positions is performed for values of a which lie within the range.

For angle approximations, the Angle absoluteθ will ultimately lie between 0o and 360o depending on the magnitude and polarity of X and Y. In order to estimate it we first estimate its primary angle primaryθ based on and as shown in Figure 32

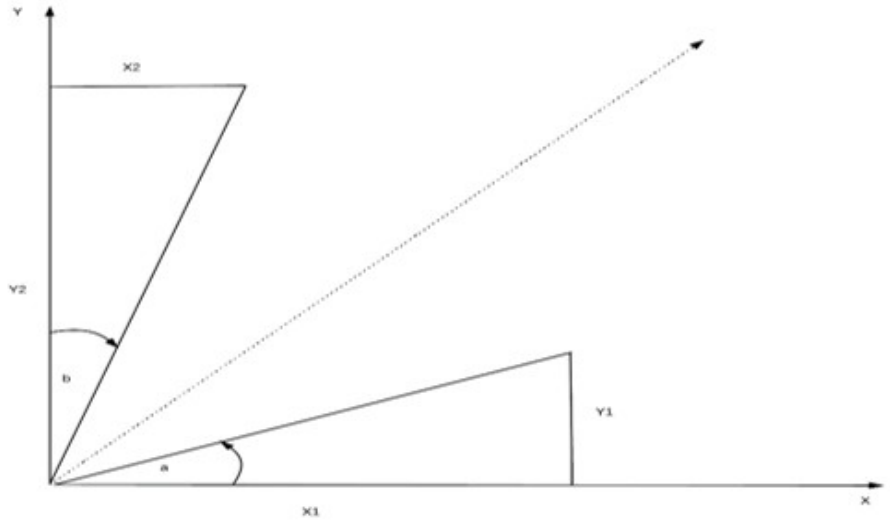

Figure 32: Primary angle calculation

$$\theta = a\ , \quad |X| > |Y|\ ;$$
$$\theta = b\ , \quad |Y| \geq |X|\ ;$$

The primary angle will always lie between 00 and 450. An accuracy of 2.50 is required so we will represent angles in the range 00 and 450 as integers 0 to 17. The ranges are selected by means of a look-up table referenced by tan (θ). The look-up table values are shown in the table below.

The absolute position represents an angle between 0° and 360°. There are 144 ranges of 2.5° between 0° and 360°. These are divided into 8 distinct areas of the chart in Figure 33.

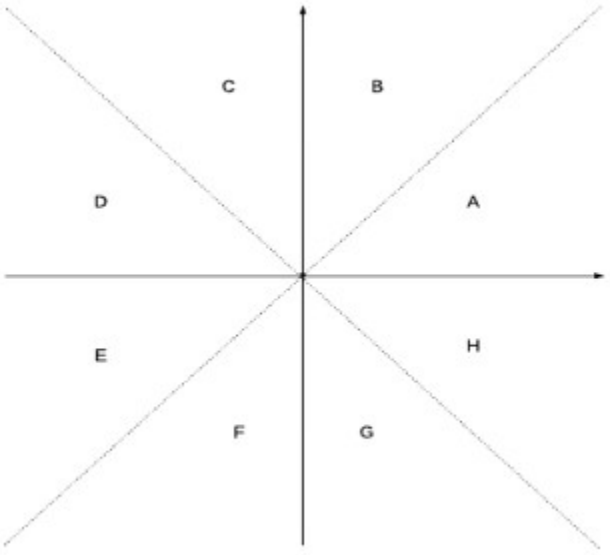

Figure 33: Regions used edge operator for arctan calculation

θ>	θ <	tan(θ) >	tan(θ)<	Position
0	2.5	0	0.043661	0
2.5	5	0.043661	0.087489	1
5	7.5	0.087489	0.131652	2
7.5	10	0.131652	0.176327	3
10	12.5	0.176327	0.221695	4
12.5	15	0.221695	0.267949	5
15	17.5	0.267949	0.315299	6
17.5	20	0.315299	0.36397	7
20	22.5	0.36397	0.414214	8
22.5	25	0.414214	0.466308	9
25	27.5	0.466308	0.520567	10
27.5	30	0.520567	0.57735	11
30	32.5	0.57735	0.63707	12
32.5	35	0.63707	0.700208	13
35	37.5	0.700208	0.767327	14
37.5	40	0.767327	0.8391	15
40	42.5	0.8391	0.916331	16
42.5	45	0.916331	1	17

In areas B,C,F,G, |X| > |Y| the while in all other areas |Y| > |X|

In areas C,D,E,F, the polarity of X is negative while in all other areas it is positive

In areas E,F,G,H, the polarity of Y is negative while in all other areas it is positive

The absolute position absoluteθ is calculated from the primary angle primaryθ as follows

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

$$absolute\theta = 35 - primary\theta, \quad |Y| > |X|, X \geq 0, Y \geq 0;$$

$$absolute\theta = 36 + primary\theta, \quad |Y| > |X|, X < 0, Y \geq 0;$$

$$absolute\theta = 71 - primary\theta, \quad |X| \geq |Y|, X < 0, Y \geq 0;$$

$$absolute\theta = 72 + primary\theta, \quad |X| \geq |Y|, X < 0, Y < 0;$$

$$absolute\theta = 107 - primary\theta, \quad |Y| > |X|, X < 0, Y < 0;$$

$$absolute\theta = 108 + primary\theta, \quad |Y| > |X|, X \geq 0, Y < 0;$$

$$absolute\theta = 143 - primary\theta, \quad |X| \geq |Y|, X \geq 0, Y < 0;$$

There are three theta Modes. In normal mode an 8 bit value is calculated as a representation of the overall angle between 0 and 360 degrees to an accuracy of 2.5 degrees (0-143) 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 degrees to 2.5 degrees accuracy is output (0-71)

In X and Y axis reflection Mode all values below the X axis are reflected in the X axis and then all values to the left of the Y axis are reflected in the Y axis. A value representing an angle between 0 and 90 degrees to 2.5 degrees accuracy is output (0-35)

Figure 34 Illustrates where each angle segment will lie after reflection in the X and Y axis.

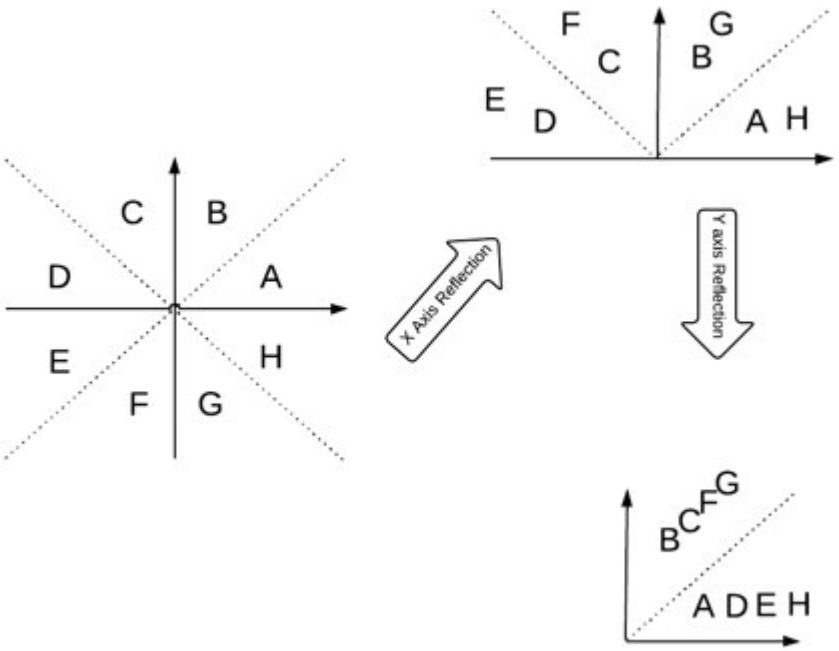

Figure 34: Transposing angle in the Theta modes

7.14.1 EdgeParam Configuration

Name	Bits	Description
frmDim		PRIVATE - frame dimensions in pixels
	15:0	frame width (must be >=**k**)
	31:16	frame height (must be >=**k**)
cfg	1:0	Input mode 00 – Normal mode U8 pixel data 01 – Precomputed FP16 (X, Y) gradients 10 – Precomputed U8 (X, Y) gradients
	4:2	Output Mode 000 – 16 bit magnitude 001 – 8 bit scaled magnitude 010 – 8 bit magnitude, 8-bit 011 – 8-bit 100 – X, Y gradients scaled to 8 bits (2's complement) 101 – X, Y gradients output in FP16
	6:5	Theta Mode 00 – Normal Theta Mode 01 – X Axis reflection 10 – X and Y axis reflection
	15:6	Bits are reserved, read-only (zero)
	31:16	Magnitude Scale Factor in fp16

Name	Bits	Description
xCoeff All in signed 2's complement format	4:0 9:5 14:10 19:15 24:20 29:25	XCoeff_a XCoeff_b XCoeff_c XCoeff_d XCoeff_e XCoeff_f
yCoeff All in signed 2's complement format.	4:0 9:5 14:10 19:15 24:20 29:25	YCoeff_a YCoeff_b YCoeff_c YCoeff_d Ycoeff_e YCoeff_f

7.15 Harris Corner detector

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_HARRIS_ID	U8	FP16, FP32	HarrisParam	9

Input	Up to 16 planes of U8
Operation	Harris corner detection with configurable kernel size (5x5, 7x7 or 9x9) and programmable *k*
Input buffer	Minimum number of lines in buffer must match programmed dimensions of filter kernel
Output	Up to 16 planes of FP16/FP32 scores
Instances	1

The Harris corners filter performs corner detection on U8F image data. The filter calculates dx and dy for all pixels in 7x7 area (i.e. radius of 3) around the current pixel using a 3x3 sub-kernel at each pixel location in the 7x7 area. The 3x3 sub-kernel used for the computation of dx is as follows:

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

For dy the transpose of this same kernel is used. The data is The terms xx, xy and yy are computed on the output of column-wise dy and dx operations accumulated for each column as the input lines stream through as follows:

- Compute dx and dy for each pixel of each column of input pixels
- Compute xx, xy and yy for each pixel of the column ans sum

- Shift the results into three 7 entry shift registers (one each for xx, xy and yy)
- When 7 columns have been received start summing the contents of the shift registers giving the final xx, xy and yy sums for one pixel. Compute det, trace, final score etc.

Please refer to the Myriad 2 datasheet for detailed operation description of the Harris kernel [4].

7.15.1 HarrisParam Configuration

Name	Bits	Description
frmDim	 15:0 31:16	PRIVATE - frame dimensions in pixels – frame width (must be >=**k**) – frame height (must be >=**k**)
cfg	3:0 4	kernel size **k**, configures width and height of pixel kernel. Valid values of **k** are 5, 7 and 9. **NOTE:** This must be configured correctly before use. The reset value of 0 is invalid. If 1, output the determinant
kValue	float	Harris corners filter K value (FP32) The k value changes the response of the edges.

7.16 Convolution (3x3 or 5x5)

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_CONV_ID	U8F, FP16	U8F, FP16	ConvParam	3, 5

Input	Up to 16 planes of FP16/U8F
Operation	FP16 precision convolution filter with configurable kernel size (3x3 or 5x5) and fully programmable FP16 coefficients. Absolute or square value of results may be selected for output. Results are conditionally (based on programmable threshold) accumulated and counted
Input buffer	Minimum number of lines in buffer must match programmed dimensions of filter kernel (3x3 or 5x5)
Output	Up to 16 planes of FP16/U8F + FP32 accumulation and U32 accumulation count. Output may be disabled if only the accumulation/count are of interest
Instances	1

The 5x5 convolution kernel is used to apply arbitrary (i.e. non-separable) convolutions. It may also be programmed to apply a 3x3 convolution. There are 25 independently programmable coefficients. The coefficients are in FP16 format.

C00	C01	C02	C03	C04
C10	C11	C12	C13	C14
C20	C21	C22	C23	C24
C30	C31	C32	C33	C34
C40	C41	C42	C43	C44

Figure 35: Labeling of kernel coefficients

Additionally, the filter can be programmed to output the absolute value or the square of the result. Finally, the output values can be accumulated into a statistics register. A threshold can be specified. Output values below this threshold are not added to the accumulator. A second statistics register is incremented every time a pixel is output that exceeds the threshold and has its value added to the accumulator. The accumulation register is FP32, and the "accumulator modified" register is U32.

7.16.1 ConvParam Configuration

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

Name	Bits	Description
frmDim	 15:0 31:16	PRIVATE - frame dimensions in pixels – frame width – frame height
cfg	2:0 3 4 5 6 7 23:8	Kernel size, **k**. Must be 3 or 5. Output clamp. Set to 1 to clamp the filter FP16 output into the range [0, 1.0] Set to 1 to output the absolute value of the result Set to 1 to output the square of the result Set to 1 to enable the accumulator Set to 1 to Disable filter output (filter can be used for stats accumulation only) Accumulation threshold – the accumulator is only updated if the output (after the absolute and square operations, if enabled) exceeds this threshold. In FP16 mode, this U8 value is scaled to the range [0, 1] before being applied.
kernel [15]	31:0	Array of FP16 coefficients. Order is C00, C01, C02, C03, C04, C10, C11 etc.

7.17 Color Combination

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_CC_ID	FP16/U8 (luma), U8 (chroma)	FP16/U8 (RGB)	CCParam	1 (Luma), 4 (Chroma)

Input	3 planes of sub-sampled chroma difference data 1 plane of FP16/U8F luma data
Operation	Upscale, de-saturate and re-combine chroma with luma to produce RGB, color-correction and saturation
Input buffers	Chroma buffer must have a minimum of 5 lines Luma buffer must have a minimum of 1 line
Output	Planar RGB in FP16/U8F
Instances	1

The Color Combination filter takes Chrominance and Luminance data that was separated from RGB previously, for the purposes of independent filtering. This filter supports:

- Upscaling of the Chrominance data back to full resolution, using an approximated Lanczos filter.
- Desaturation of the Chrominance data in dark areas.
- Combining of Chrominance and Luminance to produce RGB.
- Applying a 3x3 Color Correction and Saturation matrix to the RGB data.

The inputs to this block are:

- 4 lines of 8-bit Chroma x 3 planes (input chroma planes are subsampled by ½ in each direction, so are ¼ the size of the Luma plane).
- 1 lines of Luma (FP16 or 8-bit)
- Output is 3 planes of full-sized RGB data.

The Luma and Chroma are combined according to the following equation:

$$C_{rgb} = (L + epsilon) * C_{in} * k$$

where *k* is one of the programmed coefficients *k_r, k_g, or k_b*. The above equation is applied to all three channels (usually R, G and B). Regardless of the input format of Luma, L is converted to 0.12 Fixed Point before the above equation is applied, is normalised to the range [0, 1.0] before applying the above equation. If the chroma data was originally generated with the following equation:

$$C_{out} = C_{rgb}/(L + epsilon) \cdot (255/3)$$

Where C_{rgb} is in the range [0, 1.0] and is in the range [0, 255], then the constants *k* should be set to 3.0 (a setting of 0x300 in the **krgb1** and **krgb2** registers).

7.17.1 CCParam Configuration

This filter is configured via the CCParam:

Name	Bits	Description
frmDim	15:0 31:16	PRIVATE – frame dimensions in pixels frame width. Must be >=11 frame height. Must be >=23

Name	Bits	Description
cfg	0 2:1 15:8 23:16 25:24	Force Luma to 1 Chroma sub-sampling 0 = 4:2:0 Chroma sub-sampled horizontally and vertically 1 =4:2:2 Chroma sub-sampled horizontally 2 = 4:4:4 Chroma full size “mul” co-efficient used for Alpha computation “t1” threshold used for Alpha computation Number of planes – 1 (Value programmed is number of planes minus 1) 0 = single-plane mode, and 2 = 3-plane mode.
krgb [2]	 11:0 27:16 11:0 27:16	Coefficients for each plane (0=red, 1=green, 2=blue) “k_r” co-efficient for red plane (plane 0) for luma+chroma recombination. Format is 4.8 Fixed Point. Recommended setting = 0x300. “k_g” coefficient for green plane (plane 1) for luma+chroma recombination. Format is 4.8 Fixed Point. Recommended setting = 0x300. “k_b” co-efficient for co-efficient for blue plane (plane 2) luma+chroma recombination. Format is 4.8 Fixed Point. Recommended setting = 0x300. “epsilon” value for all planes in luma+chroma recombination. Format is 0.12 Fixed Point.
ccm [5]	 15:0 31:16	Color matrix. Format is 6.10 Signed Fixed Point. Entry [0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2],

7.18 Bayer Demosaicing Post-processing Median

Filter ID	Input Precision	Output Precision	Config Struct	Input Lines
SIPP_DBYR_PPM_ID			DbyrPpParam	3

Input	Planar RGB in up to 16 bits
Operation	De-mosaicing filter with chroma artifact removal via color channel difference,
Input buffer	Minimum of 3 lines
Output	Planar RGB in up to 16 bits
Instances	1

This kernel uses the Adaptive Homogeneity Directed (AHD) de-mosaicing algorithm with chroma artifact removal via color channel difference, 3x3 median filter and recombination.

The color differences are 16 bit signed integers. The median filters perform signed comparisons. The filtered RGB data is clamped into the range [0, 2^(data width)-1] before output.

7.18.1 DbyrPpParam Configuration

This filter is configured via the DbyrPpParam:

Name	Bits	Description
frmDim	 15:0 31:16	PRIVATE - frame dimensions in pixels – frame width. Must be >= 3 – frame height. Must be >= 3
cfg	 3:0 5:4	 Pixel width – 1 Plane multiple

8 Filter Developers 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 datasheet and the MvCV kernel library documentation.

8.1 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.

8.2 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.

8.2.1 Left/right padding and replication

In Figure 36 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 neighboring 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.

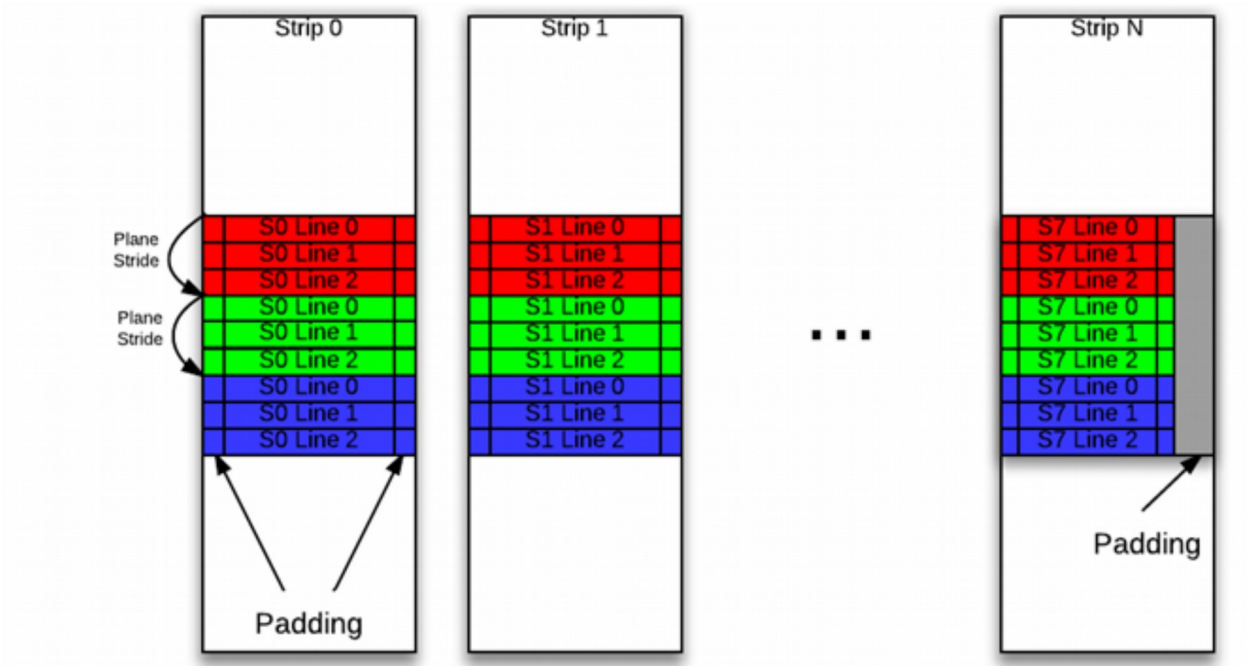

Figure 36: SIPP Buffer Padding

8.2.2 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.

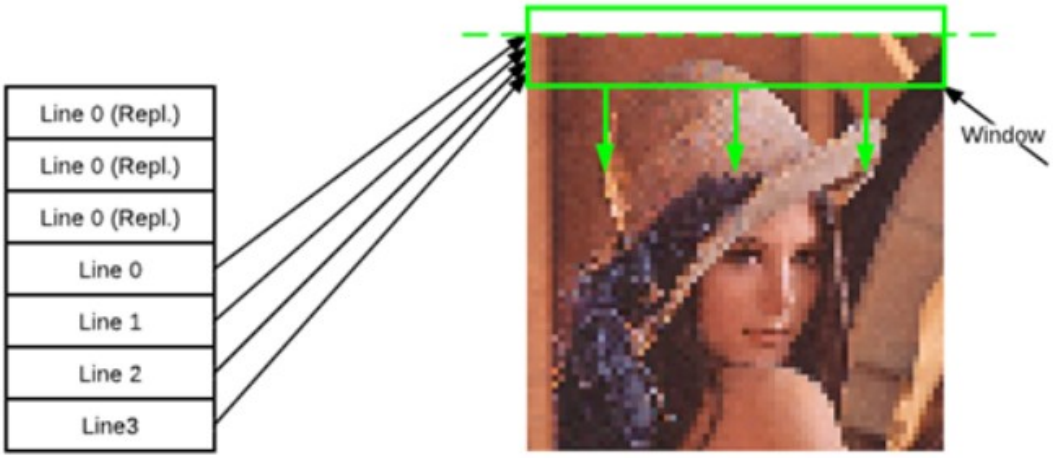

Figure 37: The SIPP framework replicates data at the top of a frame by manipulating line pointers

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.

8.3 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
}
```

8.4 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.

8.4.1 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.

8.4.2 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 10: 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 SHAVEs 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 11: The CommInfo structure

8.4.3 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 12: SIPP Argument Decorators

9 Software Filters

9.1 MvCV Kernels

The SIPP provides wrappers for some kernels in the MvCK kernel library. All SIPP kernels operate on lines of pixel elements.

The wrappers are located in common\components\sipp\filters.

Please read the MvCV documentation on the MDK install folder for details on a particular kernel.

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 optimised 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.
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

Kernel Name	Description
convNxN	Convolution optimised 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
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

Kernel Name	Description
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 pipleine.
genDnsRef	Generate reference for hardware accelerator denoise algorithm; Used in Image Signal processing pipleine.
genLuma	Generate Luminnce 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. Optimised 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)
lumaBlur	Blur opertor 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

Kernel Name	Description
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 ouput
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. Optimised 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 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.

Kernel Name	Description
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 13: SIPP Software Kernels

10 Interrupts

Generally each SIPP filter has three interrupt request sources which are mapped to three external interrupt request lines output from the SIPP and routed to the media subsystem's interrupt control block. If any filter has raised a given interrupt, and that interrupt is enabled then the corresponding external interrupt request line will be raised. (Interrupts from several filters may however be grouped together, see below for further details.) These external interrupt request lines are as follows:

sipp_irq[0] – input buffer fill level decrement interrupts
sipp_irq[1] – output buffer fill level increment interrupts
sipp_irq[2] – frame done interrupts

10.1 Input buffer fill level decrement interrupt

Filters raise their input buffer fill level decrement interrupt when they decrement their input buffer fill level(s). This interrupt is also raised in synchronous control mode even though buffer fill levels are not tracked; filters are still aware of their vertical location in the input frame and when they have completed vertical padding will raise this interrupt after every run. In buffer fill control mode this allows software to unblock the filter's *producer* if the filter is consuming lines more slowly than it is writing them to its input buffer. Generally software should track the input buffer fill level of a filter and check that there is space in the buffer before running its producer.

This interrupt is raised after the last data read from the input buffer for the current run is pushed into the filter's arithmetic data-path.

Rather than being raised every time the input buffer fill level is decremented the interrupt may be configured (via the SIPP_IBUF[N]_IR register) to trigger under one of the following conditions:

Only at the end of the frame, or

Only if the buffer fill level was decremented and the buffer was full (this event effectively un-blocks the buffer's producer), or

Only if the buffer fill level is decremented to zero and the buffer is now empty

In synchronous control mode it is not recommended to use any of the above conditions (other than the end of the frame, though that may be of limited utility) to gate the interrupt since filters do not track buffer fill levels in this configuration.

10.2 Output buffer fill level increment interrupt

In both buffer fill control mode and synchronous control mode filters will raise their output buffer fill level increment interrupt after every run. In buffer fill control mode this allows software to unblock filter's *consumer(s)* if the filter is producing lines more slowly than they are processing them. Generally software should track the output buffer fill level of a filter and check that there are sufficient lines in the buffer before running any *consumers* of this buffer.

This interrupt is raised after the last write transaction to the output buffer for the current run has been accepted by the AMC (but before the data reaches memory).

Rather than being raised after every run the interrupt may be configured (via the SIPP_OBUF[N]_IR register) to trigger under one of the following conditions:

- Only at the end of the frame, or
- Only if the buffer was empty

For the purpose of scheduling filter runs in synchronous control mode it is recommended that software use

this interrupt to determine when filters have finished their runs since it is raised on every run irrespective of vertical padding/vertical location within the frame.

10.3 Frame done interrupt

The frame done interrupt is raised by each filter when it completes processing of a frame. By default the frame done interrupt is aligned with the last output buffer fill level increment for the frame. However, it is generally possible to instead align it with the input buffer fill level decrement interrupt via the SIPP_INT2_CFG register.

There are two exceptions – the MIPI *Rx* and MIPI *Tx* filters. The MIPI *Rx* filter has no input buffer so its frame done interrupt is always aligned with the last output buffer fill level increment of the frame. The MIPI *Tx* filter has no output buffer so, by default, its frame done interrupt is aligned with the last input buffer fill level decrement for the frame. It is however possible to use other interrupt sources driven by the MIPI controller parallel interface timing (see Myriad 2 Platform Datasheet, v1.03 for further details).

10.4 Interrupt barriers

The interrupt sources driving each interrupt may be grouped via an *interrupt barrier* mask. There is one interrupt barrier mask per interrupt request line. To add an interrupt source to the barrier group the corresponding bit in the barrier mask register should be set. All interrupt sources included in the barrier group must be valid before the interrupt is requested. Any sources not included in the barrier group will cause interrupt request individually as normal. The barrier group mechanism is implemented as follows:

- The barrier mask for a given interrupt is inverted and ORed with (the barrier mask ANDed with the interrupt status register).
- A reduction OR of that gives barrier group interrupt request enable
- (However, the barrier interrupt request is only enabled if any bit is actually set in the barrier mask)
- The barrier request is ORed with the normal interrupt request (i.e. from enabled individual interrupt sources not grouped in the barrier)

Interrupt sources grouped in the barrier are masked out of the enable used for normal interrupt requests.