ACF User Guide

UG-10267-03-16

Revision History

Version	Details of Change	Authors	Date
01	Initial release	CM	June 30, 2013
02	Removed mention of ICP_Image. Misc grammer/typo fixes. Minor updates in autobuild process related to new perl scripts. Added ACF_ATTR_IN_STATIC_GLB_FIXED to replace ACF_ATTR_CFG.	CM	Sept 30, 2013
03	Minor change to required tools section.	CM	Nov 12, 2013
04	Remove version information from required tool section to keep the document more general; this information is captured elsewhere	CM	Nov 22, 2013
05	Fix documentation related to spatial dep (order should be left, right, top, bottom). Update port attribute and document advanced functionality.	CM	Feb 17, 2014
06	Misc edits.	CM	Feb 19, 2014
07	Added section on Flex license activation.	SP, AO	Aug 13, 2014
08	Fix typo in histogram kernel. Make note of limitations associated with ConnectIndrectInput. Updates to ek and sd sections. Changes related to run-time chunk size selection.	CM	Oct 23, 2014
09	ROI section added. Misc. edits and improvements.	CM	Jan 22, 2015
10	SelectApuConfiguration section added. Misc edits and improvements.	CM	Oct 22, 2015
11	Add ACF_VAR_NUM_INPUT_ITERATIONS and ACF_VAR_CU_ARRAY_WIDTH variable info. Replace ICP_ContigDataDesc with DataDescriptor Removed build details.	CM	May 2, 2016
12	Clarification added for ConnectIndirectInput	CM	Dec 12, 2016
13	Added section about interrupts	CM	May 3, 2017
14	Clean up attributes list + replace DataDescriptor with uMat + add caveat related to explicit chunk size selection + misc. edits	CM	June 27, 2017
15	Misc. edits and clarifications related to known limitations	CM	Feb 16, 2018
16	Add section for ACF offline errors	CM	Dec 10, 2018

Table of Contents

Table of Figures

Table of Tables

1 Document Purpose and Introduction

This document is an APEX Core Framework (ACF) user guide, and it contains all the information required for an ACF user to begin mapping an application (or selected parts of an application) to the APEX platform. It will outline the prerequisite software components, present a high-level design methodology, and present the necessary programming interfaces.

1.1 Acronyms

Acronym	Definition
ACF	APEX Core Framework
ACP	Array Controller Processor
APU	Array Processor Unit
CU	Computational Unit
DAG	Directed Acyclic Graph

Table 1 - Acronyms

1.2 Scope

This document is targeted towards an audience interested in accelerating applications via NXP's APEX and ACF technology. The audience should have a general familiarity with parallel processing, and a basic understanding of the APEX hardware architecture.

1.3 References

See document UG-10267-04-##-ACF_Reference_Guide.pdf for more information about ACF and ACF interfaces.

2 ACF Overview and Terminology

2.1 High Level Overview

At the highest level, ACF is an abstraction layer for the APEX hardware (HW), abstracting data movements and execution beneath a high-level interface.

The purpose of ACF is to provide a programming model and the means for a user to implement and execute common data processing tasks on the APEX without having to deal directly with the underlying hardware. While this document discusses ACF in the context of APEX HW in general, it is primarily focused on the mapping of processing tasks to the Array Processor Unit (APU).

2.2 Low Level Overview

At a slightly lower level (assuming the APU is the chosen processor), ACF is responsible for creating a processing pipeline that manages the following three steps:

1. Transferring data from external/host memory to APU memory (ACF is responsible for managing APU memory associated with the processing pipeline)
2. Process input data (residing in APU memory) with the APU processor to produce output data (also in APU memory)
3. Transfer output data from APU memory back to external/host memory

On the surface these three steps appear relatively straightforward, but things can get complicated very quickly when dealing with a SIMD array of processors (each with relatively small amounts of local memory), cascaded processing tasks with spatial dependencies, padding, etc.

Much of the complexity associated with mapping a processing scenario to the APU relates to the need for efficient data movement between external/host memory and APU memory. One of ACF's main responsibilities is to minimize the cost of such data movement. Typically, the input to a processing task is a very large amount of data, like an image or a frame of video. Minimizing the cost associated with data transfers is accomplished by:

- Pipelining *data transfers* with *processing* to hide the cost of moving data to and from APU memory.
- Combining multiple processing tasks into a single process, allowing the framework to take advantage of data locality and local intermediate results. In this way, the required input data is transferred from external memory to APU memory *once*. It is then fully processed, and the results are transferred back to external memory *once* (see the two scenarios presented in Figure 1 below). This approach significantly reduces the overhead and bandwidth associated with data movement.

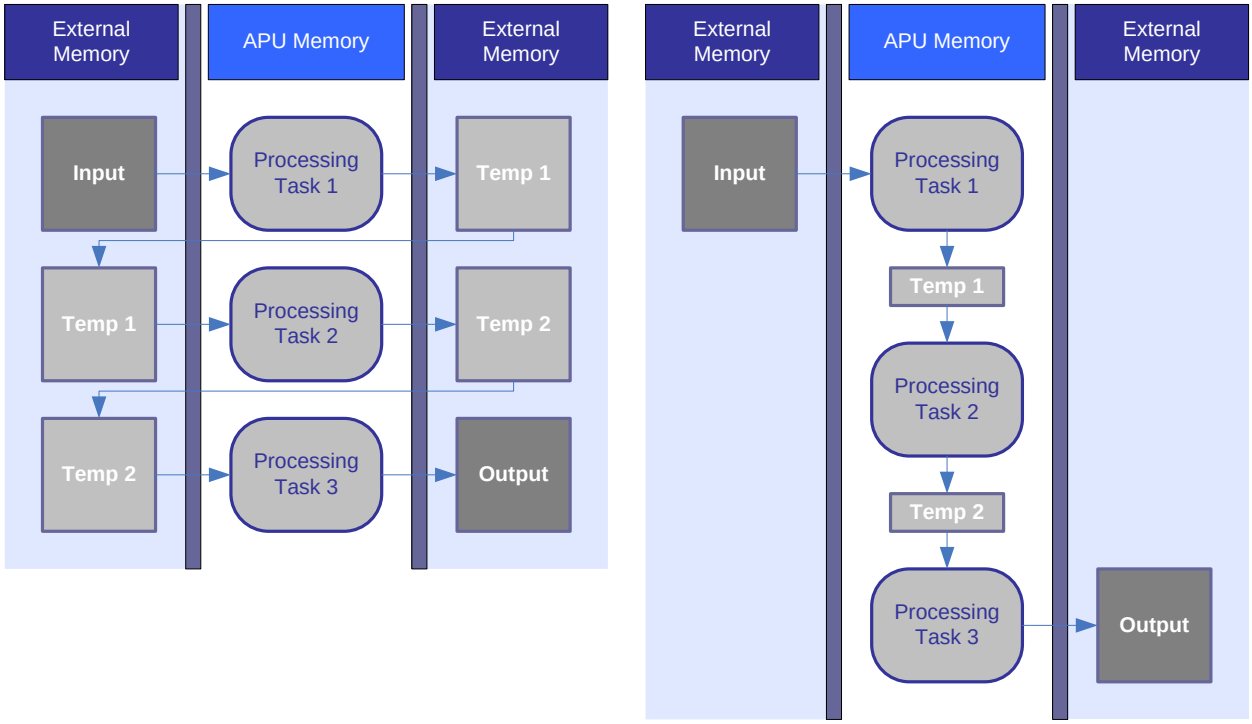


Figure 1 - Minimizing data movement between host and APU

ACF is designed to abstract the tedious and time consuming tasks associated with mapping a processing scenario to the APU. By allowing ACF to manage complex data transfers, pipelining, and sequencing, the user is free to focus on defining their processing scenario at a high level and be sure that it is mapped to the APU correctly and efficiently.

2.3 Terminology: Kernel, Graph, Process

The following concepts/definitions will be used frequently throughout this document, and it is important that their meanings be clear and unambiguous.

2.3.1 Kernel Definition

A kernel is a well-defined unit of processing that executes on a specific processor. It takes well-defined inputs, processes them, and produces well-defined outputs. Exactly what goes on inside a kernel is generally unknown to the framework (i.e. it is more or less a black box), however, interface and meta-data requirements must be adhered to by all kernels.

It is important to note that kernels are processor specific. Specific details regarding kernel authoring can be found in section 3.2.

Figure 2 - ACF Kernel

The diagram above depicts a simple addition kernel that takes two 8-bit inputs and produces one 16-bit output.

2.3.2 Graph Definition

An ACF graph is a directed acyclic graph (DAG) comprised of kernels and the directed connections between them. The information captured by a graph strictly relates to kernels and their interconnections.

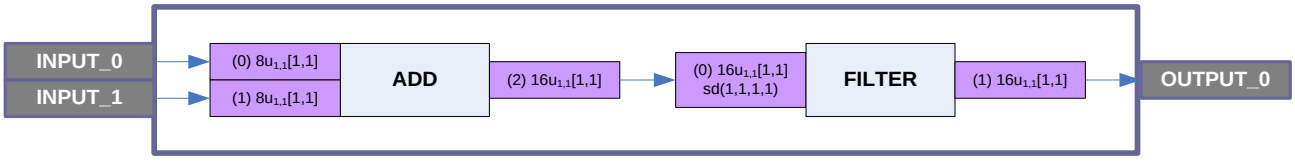


Figure 3 - ACF Graph

The diagram above depicts the same addition kernel seen in Figure 2 with the output connected to a filter kernel. Note the presence of graph-level IOs **INPUT_0**, **INPUT_1** and **OUTPUT_0** in Figure 3.

2.3.3 Process Definition

A process represents a graph that has been mapped to a processor architecture. This mapping is referred to as *resolution* (i.e. a graph was resolved to a process). In order to generate a process, a graph must be selected, a processor must be selected, and any necessary processor specific configuration information must be provided.

A process is the 'ready-to-run' form of the application/algorithm represented by a graph. In a run-time setting, a process can be loaded, configured (i.e. I/O configuration), and executed.

3 Programming with ACF

3.1 Overview

Using ACF to accelerate a processing task requires the following 5 steps:

1. Write required kernel(s) or select from pre-existing kernel(s) and/or a kernel library.
2. Construct a graph using desired kernels by specifying connections between them
3. Create a process description that links the graph created in step 2 to the APU, and provide any necessary processor specific configuration.
4. Use the auto-build script to resolve the process description created in step 3; this produces the final ACF outputs (i.e. process binary and C++ object encapsulating the process) needed for host-side execution.
5. Write host-side code to configure and execute the APU process created in step 4 (i.e. configure inputs and outputs, start execution, wait for completion). This code then becomes part of the host-side application and must be compiled and linked into the final library/binary that will run on the host processor.

Note that steps one through four are performed ‘offline’ in a PC environment. Step five is performed at run-time in a host processor environment.

The following sections will discuss each of these five steps in detail.

3.2 Writing APU Kernels (Step 1)

An APU kernel is a unit of processing meant to execute on the APU. Kernels must be written in adherence to a set of rules related to kernel interface and port specification. A kernel description typically consists of three parts:

1) **Kernel implementation**: this is the kernel implementation in APU-C (i.e. C99 code with vector extensions)
2) **Kernel metadata**: this is information that uniquely identifies the kernel and characterizes kernel inputs and outputs (referred to as 'ports'). Most of this information is quite general and makes sense for a wide variety of kernels. Filling in this information for a kernel is a good way to determine if a kernel will comfortably fit within the ACF framework. It is critical that the information provided in the port specification section accurately reflect kernel I/O requirements.
3) **Kernel wrapper for ACF**: this is the method that wraps the kernel implementation so it can be used by ACF.

3.2.1 Kernel Implementation

Figure 4 below presents an example implementation for an addition kernel 'ADD'.

add_implementation.h

```
#ifndef _ADD_IMPLEMENTATION_H
#define _ADD_IMPLEMENTATION_H

#include <stdint.h>

void ADD (vec08u* lpvIn0, int16_t lStrideIn0,
          vec08u* lpvIn1, int16_t lStrideIn1,
          vec16u* lpvOut0, int16_t lStrideOut0,
          int16_t lChunkWidth, int16_t lChunkHeight);

#endif //_ADD_IMPLEMENTATION_H
```

add_implementation.c

```
#ifdef ACF_KERNEL_IMPLEMENTATION

#include "add_implementation.h"

void ADD (vec08u* lpvIn0, int16_t lStrideIn0,
          vec08u* lpvIn1, int16_t lStrideIn1,
          vec16u* lpvOut0, int16_t lStrideOut0,
          int16_t lChunkWidth, int16_t lChunkHeight)
{
   for (int16_t y=0; y<lChunkHeight; y++)
   {
      for (int16_t x=0; x<lChunkWidth; x++)
      {
         lpvOut0[y*lStrideOut0+x] = (vec16u)lpvIn0[y*lStrideIn0+x] +
                                    (vec16u)lpvIn1[y*lStrideIn1+x];
      }
   }
}

#endif //#ifdef ACF_KERNEL_IMPLEMENTATION
```

Figure 4 - Example ADD kernel implementation

For maximum flexibility, kernels should be written with variable processing loops that are inputs to the kernel. In this example a processing loop is set up based on the *lChunkWidth* and *lChunkHeight* input parameters. 'Chunk' simply refers to the 1D or 2D region of data to be processed by the kernel.

```
for (int16_t y=0; y<lChunkHeight; y++)
{
   for (int16_t x=0; x<lChunkWidth; x++)
   {
      <core kernel processing goes here!>
   }
}
```

It is required that the kernel implementations always make use of the chunk width, chunk height, and stride information when setting up processing loops. These are input parameters provided to the kernel by the framework and ACF is free to select values for these parameters to satisfy the processing pipeline requirements.

Also note that kernel inputs must always be treated as read only (i.e. a kernel should never write back to a port it has defined as an input).

The core processing of the ADD kernel is simply an addition of the two inputs to produce one output.

```
lpvOut0[y*lStrideOut0+x] =
          (vec16u)(lpvIn0[y*lStrideIn0+x] + lpvIn1[y*lStrideIn1+x]);
```

3.2.2 Kernel Metadata

Figure 5 (below) depicts the metadata and ACF wrapper for the ADD kernel.

add_wrapped_for_acf.c

```
#ifdef ACF_KERNEL_METADATA

static  KERNEL_INFO _kernel_info_add
(
   "ADD",
   3,
   __port(__index(0),
          __identifier("INPUT_0"),
          __attributes(ACF_ATTR_VEC_IN),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d08u),
          __e0_size(1, 1),
          __ek_size(1, 1)),
   __port(__index(1),
          __identifier("INPUT_1"),
          __attributes(ACF_ATTR_VEC_IN),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d08u),
          __e0_size(1, 1),
          __ek_size(1, 1)),
   __port(__index(2),
          __identifier("OUTPUT_0"),
          __attributes(ACF_ATTR_VEC_OUT),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d16u),
          __e0_size(1, 1),
          __ek_size(1, 1))
);

#endif //#ifdef ACF_KERNEL_METADATA

#ifdef ACF_KERNEL_IMPLEMENTATION

#include "add_implementation.h"

void ADD (kernel_io_desc lIn0, kernel_io_desc lIn1, kernel_io_desc lOut0)
{
   vec08u* lpvIn0  = (vec08u*)lIn0.pMem;
   vec08u* lpvIn1  = (vec08u*)lIn1.pMem;
   vec16u* lpvOut0 = (vec16u*)lOut0.pMem;

   ADD(lpvIn0, lIn0.chunkSpan,
       lpvIn1, lIn1.chunkSpan,
       lpvOut0, lOut0.chunkSpan/2,
       lIn0.chunkWidth, lIn0.chunkHeight);
}

#endif //#ifdef ACF_KERNEL_IMPLEMENTATION
```

Figure 5 - ADD kernel metadata and ACF wrapper

Note that this file includes a metadata section at the top, and the kernel wrapper method ‘ADD’ beneath the metadata. The #defines surrounding each section (i.e. `ACF_KERNEL_METADATA` and `ACF_KERNEL_IMPLEMENTATION` respectively) are required and it is currently advised to put metadata and wrapper method in the same file for each kernel as shown in the example above.

The first field in the metadata for the ADD kernel is the kernel identifier “ADD”; this identifier is important because it will be used later to refer to this kernel when creating a graph. This identifier must be unique as it is the only kernel ‘handle’ that exists and it must not clash with another kernel identifier. Kernel identifier length should not exceed 64 characters.

The second field contains the number of ports, which must correspond to the number of parameters in the kernel function signature. The ADD kernel has 3 ports.

Next are the descriptions for each of the three kernel ports. For each port (i.e. each input/output), a set of characteristics must be provided. Table 2 (below) outlines the various port characteristics.

Characteristic	Description
__index	The index of the associated parameter in the kernel function signature. This index links a conceptual *port* to a concrete *function parameter*. For example, the port characterized with _index(0) in Figure 5 describes the first parameter `lIn0` in the kernel function signature. Likewise, the port characterized with _index(1) describes the second parameter `lIn1`, etc. The maximum number of ports for a single kernel should not exceed 32. Usage: __index(<kernel parameter index starting from 0>) Example : `__index(0)`
__identifier	A string-based identifier that will be used to identify and refer to the port during graph creation. Port identifier length should not exceed 64 characters. Usage : __identifier(<port identifier string>) Example : `__identifier("INPUT_0")`

__attributes	This characteristic is responsible for relaying details about the port type to the framework. See section 3.2.2.1 for an explanation of the port attribute nomenclature.
	Possible values:
	Vector input types:
	`ACF_ATTR_VEC_IN`
	`ACF_ATTR_VEC_IN_FIXED`
	`ACF_ATTR_VEC_IN_STATIC`
	`ACF_ATTR_VEC_IN_STATIC_FIXED`
	Vector output types:
	`ACF_ATTR_VEC_OUT`
	`ACF_ATTR_VEC_OUT_FIXED`
	`ACF_ATTR_VEC_OUT_STATIC`
	`ACF_ATTR_VEC_OUT_STATIC_FIXED`

	Scalar input types:
	`ACF_ATTR_SCL_IN`
	`ACF_ATTR_SCL_IN_FIXED`
	`ACF_ATTR_SCL_IN_STATIC`
	`ACF_ATTR_SCL_IN_STATIC_FIXED`
	
	Scalar output types:
	`ACF_ATTR_SCL_OUT`
	`ACF_ATTR_SCL_OUT_FIXED`
	`ACF_ATTR_SCL_OUT_STATIC`
	`ACF_ATTR_SCL_OUT_STATIC_FIXED`
	
	Usage :
	__ attributes (<attribute>)
	Example :
	`__attributes(ACF_ATTR_VEC_IN)`

__spatial_dep	Specifies input spatial data dependencies (in units of e0) to the left, to the right, above, and below assuming a 2D data organization (dependencies need not be symmetrical). **The framework performs replication padding for input *border* padding as required**. Note that this characteristic is **only applicable to 2D, non-static, direct, vector inputs**. See section 3.2.2.3 for more information about spatial dependencies. Usage: __spatial_dep(<left>, <right>, <top>, <bottom>) Example: `__spatial_dep(1,1,1,1)`
__e0_data_type	Specifies the data type of element <0> (e_0). See section 3.2.2.2.1 for more information about element<0>. Possible values: • d08u – unsigned 8-bit data • d08s – signed 8-bit data • d16u – unsigned 16-bit data • d16s – signed 16-bit data • d32u – unsigned 32-bit data • d32s – signed 32-bit data Usage: __e0_data_type (<data type>) Example: `__e0_data_type(d08u)`

__e0_size	Specifies the size of element<0> (e_0). See section 3.2.2.2.1 for more information about element<0>. Usage: __element_0(<width>, <height>) Example: `__element_0(1,1)`
__ek_size	Specifies the size of element <k> (e_k). See section 3.2.2.2.2 for more information about element<k>. Usage: __element_k(<width>, <height>) Example: `__element_k(1,1)`

Table 2 - Kernel port characteristics

Based on the port specification in Figure 5, it should be clear that the ADD kernel has two 8-bit unsigned input ports and one 16-bit unsigned output port. None of the ports have spatial dependencies. The smallest unit of input data the kernel can operate on is a single 8-bit value (dictated by __e0_data_type, __e0_dim, and __ek_dim).

3.2.2.1 **Port Attribute Nomenclature**

Port attribute definition follows a strict nomenclature comprised of various keywords, and this section will provide a means to translate and interpret this nomenclature. Note that while this nomenclature strives to be as general as possible by design, specific details related to the APEX architecture and the ACF processing model will be provided for more advanced, architecturally aware port types for the purpose of improved clarity

- **IN / OUT**
  - This port attribute is always explicitly expressed and indicates if a port is an input port (IN) or an output port (OUT).
- **VEC / SCL**
  - This port attribute is always explicitly expressed and indicates whether data should be associated with vector or scalar memory.
  - **VEC** – Vector data will be distributed across or read from the local memories of the processors that comprise the SIMD vector processing array in the APU. From a kernel point of view, data associated with a vector port should be interpreted as vector data (e.g. vec08u, vec16u, vec32u, etc.).
  - **SCL** - Scalar data will be written to or read from the local memory of the ACP processor in the APU. From a kernel point of view, data associated with a scalar port should be interpreted as scalar data (e.g. int8_t, int16_t, int32_t, etc.).
- **STATIC / (non-static)**

  - The **STATIC** port attribute indicates that there will only be a single instance of the memory associated with the port data, and that the framework will treat the memory associated with this port as monolithic and persistent during pipeline execution.
  - If the STATIC port attribute is not specified, it is assumed the memory associated with the port is NOT static. In this case the framework is free to allocate memory to meet the requirements of the processing pipeline (e.g. n-degree buffering, circular buffering, etc.).
- **FIXED / (non-fixed)**
  - The **FIXED** port attribute indicates that the size of the data is specified *exactly* by __ek_dim (in units of e0) and shall not be scaled in any way by the framework.
  - If the FIXED port attribute is not specified, it is assumed that the size of the data associated with the port is NOT fixed, and the framework is free to scale the size of the data being processed (based on the guidelines set by __ek_dim) to coincide with the optimal processing pipeline.
  - Example: use a FIXED output port when kernel output size has no meaningful dependency on kernel input size. For example, consider a kernel written to process a chunk of input data and output a single 32-bit value that contains the sum of all the values in the input chunk. In such a use case, no matter the size of the input data (8x1, 4x4, 8x8, etc.), the output is always a single 32-bit value, and should therefore be specified as FIXED.

For more information about port types and for a discussion of more advanced use cases, please see section 3.7.

3.2.2.2 **Element<0> (e_0) and Element<k> (e_K)**

The 'element' nomenclature exists to allow maximum flexibility when expressing the kind of data a kernel I/O can handle. The two element types can be seen as a hierarchy where e_0 is the base data type and e_k is an array of e_0's.

3.2.2.2.1 Element<0>

Element<0> (or e_0) represents the smallest meaningful data granularity for a kernel I/O. For an 8-bit grayscale image this would be a single byte. For a packed/interleaved YUV422 image this would be a YUYV sample 'pair'.

Let e_0 be written as:

e_0 = <element type>$_{<\text{num element in x dim}>,<\text{num elements in y dim}>}$

where 'element type' can be 8u, 8s, 16u, 16s, 32u, or 32s.

Examples:

If your element is a single unsigned byte $e_0 = 8u_{1,1}$

If your element is an 8x1 array of signed 8-bit values $e_0 = 8s_{8,1}$

If your element is a 4x1 array of unsigned 16-bit values $e_0 = 16u_{4,1}$

If your element is a 2x2 array of unsigned 8-bit values $e_0 = 8u_{2,2}$

e_0 is important because it is used for 'type checking' when trying to connect kernels and I/Os. For example, if e_0 specified by the output port of kernel A does not match e_0 specified by the input port of kernel B, a connection cannot be made between these two ports.

3.2.2.2.2 Element<k>

Element<k> (or e_k) is meant to express the smallest 2D array of e_0's that make sense for a kernel IO ***based on the kernel implementation***.

Let e_k be written as:

$e_k = e_0$ [<num e_0 in x dim>,<num e_0 in y dim>]

Examples:

If the smallest unit of data a kernel can operate on is a single unsigned 8-bit value (i.e. $e_0 = 8u_{1,1}$) and there are no additional kernel-implementation related restrictions, e_k will be '1' in both the x and y dimensions. e_k=[1,1] is the most common case:

$e_k = e_0$ [1,1] = $8u_{1,1}$ [1,1]

If a kernel operates on unsigned 16-bit data (i.e. $e_0 = 16u_{1,1}$) but the kernel implementation requires a 2x2 array of e_0's:

$e_k = e_0$ [2,2] = $16u_{1,1}$ [2,2]

If the smallest unit of data a kernel can operate on is a is a 4x1 array of 8-bit signed values (i.e. $e_0 = 8s_{4,1}$) and the kernel implementation requires a 2x1 array of e_0's:

$e_k = e_0$ [2,1] = $8s_{4,1}$ [2,1]

When possible, always try to write kernels with $e_k=e_0$[1,1]. This gives the kernel more flexibility and allows the framework to use the kernel in a wider variety of circumstances. In most cases, e_k will naturally be [1,1] since most kernel implementations don't impose restrictions on the smallest unit of processing beyond that implied by e_0.

***Note that spatial dependencies should not be considered when defining e_k.** It is completely valid to have (for example) $e_k=8u_{1,1}$ [1,1] and sd=(5,5,5,5), since e_k and sd express different things.

e_k is especially important in kernels that deal with data rate changes. In addition to characterizing the smallest chunk of data that can be accepted by a kernel I/O, e_k can express data rate changes that may occur between kernel input and output. Consider a kernel that decimates an input by 2 in the x and y directions. It doesn't make sense for this kernel to have an input $e_k = 8u_{1,1}$ [1,1] because such an input cannot be decimated (it is just a single 8-bit value). Instead, the kernel I/O should be expressed as follows:

Figure 6 - Decimate kernel (e_k)

By specifying e_k=[2,2] for the input, it ensures that the kernel always receives *at least* a 2x2 chunk of e_0's at the input port. The difference between input and output e_k's make it clear that a data rate change has occurred.

3.2.2.3 **Kernels and Spatial Dependencies**

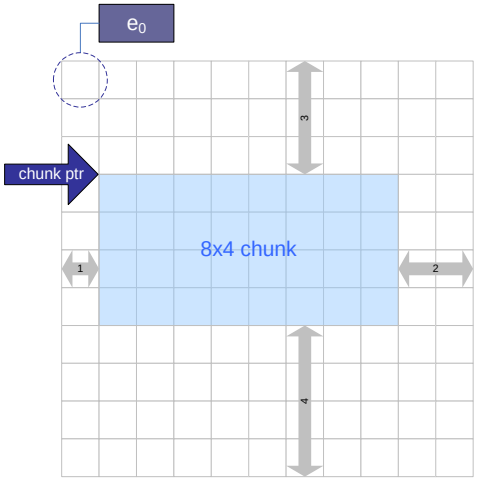
Spatial dependencies can be expressed for 2D non-static vector inputs. By allowing a kernel developer to express spatial dependencies, it allows him/her to write a more generalized kernel that operates on an input chunk with flexible dimensions.

Spatial dependency information is expressed as an array of 4 values as follows:

sd (<dep_{left}>, <dep_{right}>, <dep_{top}>, <dep_{bottom}>)

Note that this 'sd' shorthand notation corresponds to the metadata port characteristic '__spatial_dep' in Table 2 and it should only be specified for input ports.

By specifying a spatial dependency on an input, the framework is being told that it must make data *beyond* chunk boundaries locally available to the kernel for processing. For example, assume an 8x4 chunk of data is fed into a kernel that specifies sd (1,2,3,4). In this scenario, the framework will invoke the kernel on a region of memory that resembles the following:



***If a chunk coincides with an input edge/border **ACF performs replication padding (e0 resolution)** for applicable edges. ACF will generate top edge padding for chunks that coincide with the top edge of an input, replicating the first line of the chunk. Similarly, left edge padding will be generated for chunks that coincide with the left edge of an input, replicating the first column of the chunk. Corner replication is based on the associated corner value (e.g. the value in the top left corner of the chunk is replicated to fill in the top left padding region).

Dependencies are expressed in units of e_0. A 3x3 filter would express spatial dependencies as sd (1,1,1,1). A 5x5 filter would express spatial dependencies as sd (2,2,2,2).

Referring to the diagram above, the Sobel 3x3 filter would be fully characterized as follows:

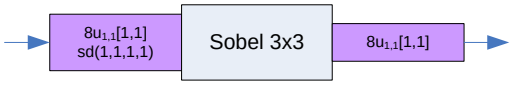


Figure 7 - Sobel 3x3 kernel with spatial dependencies

3.2.3 Kernel ACF Wrapper

Figure 5 in section 3.2.2 depicts the ACF wrapper for the ADD kernel implementation (it can be found below the metadata).

The ACF wrapper function signature must adhere to the following template:

```
void <kernel_name> (kernel_io_desc <param0>, kernel_io_desc <param1>, … )
{
   //kernel implementation
}
```

The wrapper must be a `void` function with a function name (`kernel_name`) that matches the identifier expressed in the first field of the kernel metadata associated with this kernel (in this case it is 'ADD').

It must have a parameter list of type `kernel_io_desc`, where conceptually, each parameter corresponds to a kernel port. `kernel_io_desc` is a simple descriptor that describes the *chunk* of data associated with the port; it contains the address of the data in memory, in addition to a description of the data chunk (chunkWidth, chunkHeight, and chunkSpan). It is defined as follows:

```
typedef struct _kernel_io_desc
{
   void* pMem;         //pointer to the chunk of data
   int   chunkWidth;   //width of the chunk in units of e0
   int   chunkHeight; //height of the chunk in units of e0
   int   chunkSpan;    //number of bytes to skip to get to the next line of bytes
} kernel_io_desc;
```

The typical first step in wrapping any kernel implementation is to 'unpack' the relevant address and chunk size information from each parameter/port `kernel_io_desc` structure. This structure allows access to the input and output data pointers, in addition to the necessary chunk size and span information needed for setting up processing loops. In the ADD example the unpacking is done as follows:

```
vec08u* lpvIn0  = (vec08u*)lIn0.pMem;
vec08u* lpvIn1  = (vec08u*)lIn1.pMem;
vec16u* lpvOut0 = (vec16u*)lOut0.pMem;

ADD(lpvIn0, lIn0.chunkSpan,
    lpvIn1, lIn1.chunkSpan,
    lpvOut0, lOut0.chunkSpan/2,
    lIn0.chunkWidth, lIn0.chunkHeight);
```

A few notes about this 'unpacking' step:

- Ports specified as ACF_ATTR_VEC_IN and ACF_ATTR_VEC_OUT must be cast to the appropriate ***vector*** type before use. In the above example this is seen here:

```
const vec08u* lpvIn0  = (const vec08u*)lIn0.pMem;
const vec08u* lpvIn1  = (const vec08u*)lIn1.pMem;
vec16u* lpvOut0 = (vec16u*)lOut0.pMem;
```

- Ports specified as ACF_ATTR_IN_SCL should be cast to the appropriate ***scalar*** type before use. See the filter kernel example in section 3.2.4 for an example.
- The stride of lOut0 above is calculated by dividing the chunk span by 2; this is because port 2 is associated with 16-bit data and ***span is always in bytes***.

3.2.4 Filter Kernel Example (metadata and ACF wrapper)

Figure 8 (below) depicts the metadata and ACF wrapper for the FILTER kernel.

filter_wrapped_for_acf.c

```
#ifdef ACF_KERNEL_METADATA

static  KERNEL_INFO _kernel_info_filter
(
   "FILTER",
   3,
   __port(__index(0),
          __identifier("INPUT_0"),
          __attributes(ACF_ATTR_VEC_IN),
          __spatial_dep(1,1,1,1),
          __e0_data_type(d08u),
          __e0_size(1, 1),
          __ek_size(1, 1)),
   __port(__index(1),
          __identifier("INPUT_COEF"),
          __attributes(ACF_ATTR_SCL_IN_STATIC_FIXED),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d08u),
          __e0_size(1, 1),
          __ek_size(9, 1)),
   __port(__index(2),
          __identifier("OUTPUT_0"),
          __attributes(ACF_ATTR_VEC_OUT),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d08u),
          __e0_size(1, 1),
          __ek_size(1, 1))
);

#endif //#ifdef ACF_KERNEL_METADATA

#ifdef ACF_KERNEL_IMPLEMENTATION

#include "filter_implementation.h"

void FILTER (kernel_io_desc lIn0, kernel_io_desc lInCoef, kernel_io_desc lOut0)
{
   const vec08u*  lpvIn0   = (const vec08u*) lIn0.pMem;
   const uint8_t* lpInCoef = (const uint8_t*) lInCoef.pMem;
   vec08u*        lpvOut0  = (vec08u*) lOut0.pMem;

   FILTER(lpvIn0, lIn0.chunkSpan,
          lpInCoef,
          lpvOut0, lOut0.chunkSpan,
          lIn0.chunkWidth, lIn0.chunkHeight);
}

#endif //#ifdef ACF_KERNEL_IMPLEMENTATION
```

Figure 8 - FILTER kernel metadata and ACF wrapper

Notable metadata differences compared to the previously discussed ADD kernel include:

- port INPUT_0 specifies a non-zero spatial dependency : `__spatial_dep(1,1,1,1)`

- port INPUT_COEF specifies an ACF_ATTR_SCL_IN_STATIC_FIXED port type that allows the kernel to be configured with a 9-byte coefficient array ($e_k = 8u_{1,1}$ [9,1]).

Also note the following difference in the 'unpacking' stage of the implementation:

```
const vec08u* lpvIn0   = (const vec08u*)lIn0.pMem;
const uint8_t* lpInCoef = (const uint8_t*)lInCoef.pMem;
vec08u* lpvOut0  = (vec08u*)lOut0.pMem;
```

The ACF_ATTR_VEC_IN and ACF_ATTR_VEC_OUT ports are cast to 8-bit vector types as seen in the ADD example, whereas the ACF_ATTR_SCL_IN_STATIC_FIXED port input is cast to an 8-bit scalar type.

3.3 Creating a Graph (Step 2)

Once a set of kernels is available, graph construction is a simple matter of deciding which kernels to use and how to connect them.

3.3.1 Creating a Graph Diagram

If a graph is complicated or involves multiple kernels, it is a good idea to quickly create a graph diagram. In this example, a graph will be created that uses the ADD and FILTER kernels discussed in the previous section.

From an illustration point of view, the ADD and FILTER kernels can be expressed as follows:

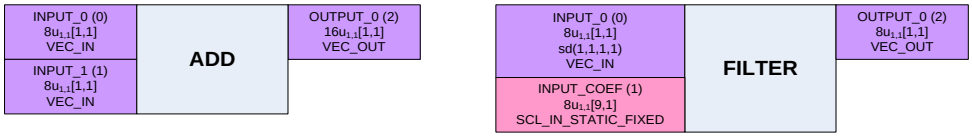


Figure 9 - ADD and FILTER kernel diagrams

The kernel diagrams above capture port details for each kernel that are relevant to graph construction. As seen in Figure 9 above, each port expresses the identifier, index, e_k, and spatial dependency information (if spatial dependency information is absent from a port it is assumed to zero). The port details in the diagrams above are simply restatements of the information expressed by the kernel metadata (refer to section 3.2.2).

Once each kernel is clearly expressed, the next step is to create a graph diagram that specifies graph-level ports and all desired connections.

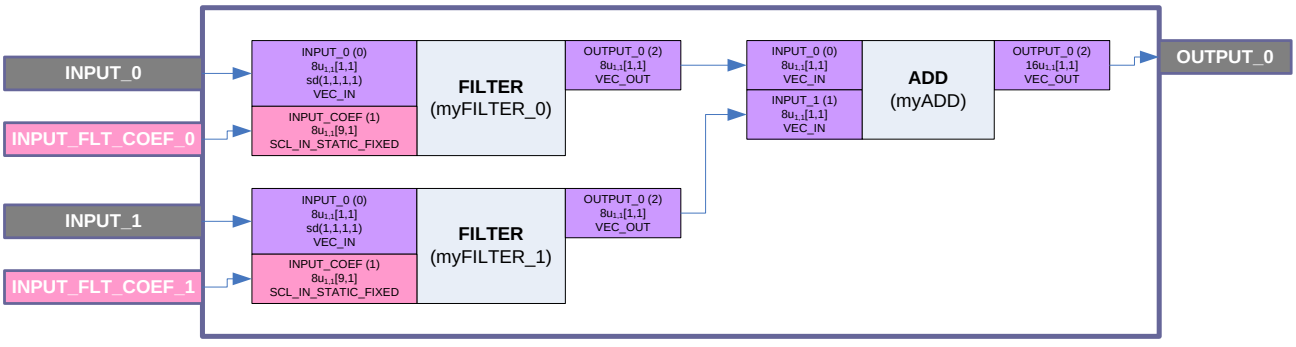


Figure 10 - A graph diagram (processingTaskA)

The graph diagram in Figure 10 makes it clear that two inputs (INPUT_0 and INPUT_1) are being filtered (the filters have configurable coefficients) and then added together to produce a single output (OUTPUT_0).

Note that five graph-level ports have been specified:

- INPUT_0
- INPUT_FLT_COEF_0
- INPUT_1
- INPUT_FLT_COEF_1
- OUTPUT_0

Graph-level ports are important because they represent the ports that will be configured in future steps (i.e. process description and host-side configuration).

3.3.2 ACF Graph and Example

Once a graph diagram exists, expressing the graph in a form that ACF understands is a straightforward exercise.

The first step is to create a *.hpp file (e.g. myGraph_graph.hpp) based on the following template:

```
#include <ACF_Graph.hpp>

class <graph class name> : public ACF_Graph
{
public:

   void Create()
   {
      //set identifier for graph
      SetIdentifier(<graph name>);

      //add kernels
      AddKernel(<local kernel identifier 0>, <kernel identifier 0>);
      AddKernel(<local kernel identifier 1>, <kernel identifier 1>);
      ...

      //add graph ports
      AddInputPort(<graph level input port 0>);
      AddInputPort(<graph level input port 1>);
      ...
      AddOutputPort(<graph level output port 0>);
      AddOutputPort(<graph level output port 1>);
      ...

      //specify connections
      Connect(<port 0>, <port 1>);
      Connect(<port 2>, <port 3>);
      ...
   }
};
```

Next, give the graph class a meaningful name and fill in the Create() method such that all kernels, ports, and connections are properly expressed. The final ACF-ready graph code that represents the example in Figure 10 is as follows:

```
#include <ACF_Graph.hpp>

class myGraph : public ACF_Graph
{
public:

   void Create()
   {
      //set identifier for graph
      SetIdentifier("myGraph");

      //add kernels
      AddKernel("myADD", "ADD");
      AddKernel("myFILTER_0", "FILTER");
      AddKernel("myFILTER_1", "FILTER");

      //add graph ports
      AddInputPort("INPUT_0");
      AddInputPort("INPUT_1");
      AddInputPort("INPUT_FLT_COEF_0");
      AddInputPort("INPUT_FLT_COEF_1");
      AddOutputPort("OUTPUT_0");

      //specify connections
      Connect(GraphPort("INPUT_0"),          KernelPort("myFILTER_0", "INPUT_0"));
      Connect(GraphPort("INPUT_FLT_COEF_0"), KernelPort("myFILTER_0", "INPUT_COEF"));
      Connect(GraphPort("INPUT_1"),          KernelPort("myFILTER_1", "INPUT_0"));
      Connect(GraphPort("INPUT_FLT_COEF_1"), KernelPort("myFILTER_1", "INPUT_COEF"));

      Connect(KernelPort("myFILTER_0", "OUTPUT_0"), KernelPort("myADD", "INPUT_0"));
      Connect(KernelPort("myFILTER_1", "OUTPUT_0"), KernelPort("myADD", "INPUT_1"));

      Connect(KernelPort("myADD", "OUTPUT_0"), GraphPort("OUTPUT_0"));
   }
};
```

The AddKernel(…) method takes two identifiers; the first is the identifier that is used throughout the graph specification to refer to that specific instance of the kernel, and the second is the unique kernel identifier specified in the kernel metadata. The first identifier is essentially a handle on a kernel instance. For example, 'myFILTER_0' is a handle on the first instance of the 'FILTER' kernel, and 'myFILTER_1' is a handle on the second instance of the 'FILTER' kernel. **If you use the same kernel multiple times in a graph, you must add multiple instances of that kernel to the graph, each with a unique local identifier**.

For more detailed explanations of the various graph construction methods, see the ACF_Graph section of UG-10267-04.

3.3.3 Known caveats and limitations

1) The maximum number of kernels allowed per graph is 100.
2) The maximum number of graph inputs is 100. The maximum number of graph outputs is 100.
3) A source port can be connected to a maximum of 100 destination ports in the case of multiple forward connections (e.g. kernel A output is connected to both kernel B input and kernel C input).
4) If the only non-fixed graph input(s) are scalar, you must either fix them in kernel metadata or fix them via the method described in section 3.4.2. ACF host run-time will not be able to perform scenario selection in this case and will return an error message indicating that a suitable scenario selection port could not be found.

3.4 Specifying a Process Description (Step 3)

The purpose of a process description is to link a graph to a specific processor, and allow for the provision of any processor specific configuration that may be required prior to resolution.

3.4.1 ACF Process Description and Example

The first step is to create a *.hpp file (e.g. myProcess_proc_desc.hpp) based on the following template:

```
#include <ACF_Process_Desc_APU.hpp>
#include "<*.hpp graph file created in step 2>"

class <process descriptor class name> : public ACF_Process_Desc_APU
{
public:

   void Create()
   {
      Initialize(mGraph, <process identifier>);
   }

   <graph class specified in graph *.hpp file> mGraph;
};
```

Notes about this template:

- ACF currently only supports the mapping of graphs to the APU processor, and **the selection of the APU as the processor is done by deriving the process descriptor class from ACF_Process_APU**, as seen above.
- The class needs to contain a member ('mGraph' in this case) that is an instantiation of the graph to be mapped to the APU.
- The 'Initialize(…)' method must be called with the graph object (e.g. mGraph) and a user-defined process identifier. **This identifier is important because it will be picked up by the build system and be used as a base handle/prefix for all generated output** associated with the final resolved process.

Filling in the template to map the graph specified in Figure 10 to the APU processor results in the following:

```
#include <ACF_Process_Desc_APU.hpp>
#include "myGraph_graph.hpp"

class myProcess_apu_process_desc : public ACF_Process_Desc_APU
{
public:

   void Create()
   {
      Initialize(mGraph, "myProcess");
   }
```

```
   myGraph mGraph;
};
```

For more detailed explanations of the various process description methods, see the ACF_Process_Desc_APU section of UG-10267-04.

3.4.2 Explicit Offline Chunk Size Specification (*ADVANCED*)

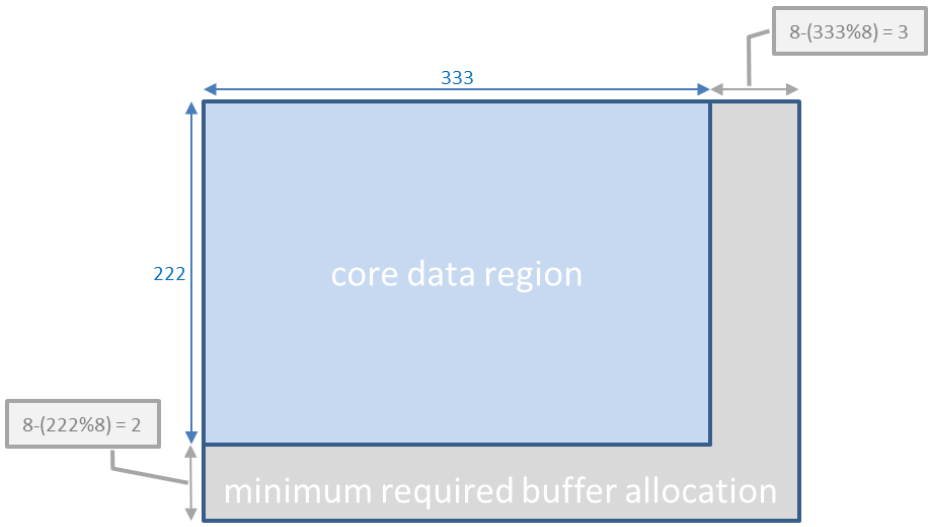
Process chunk size information is normally abstracted/hidden from the user; chunk size selection requires awareness of APU specific details and a clear understanding of how the data is broken down for processing. In most circumstances, this selection is best left to ACF at run-time.

In specific cases, it may be necessary to pre-select and 'hard code' the chunk sizes associated with non-fixed process inputs (for example, if you know you will be using the process to only deal with I/O's of a specific size and you know you will be running the process on a specific APU configuration).

!!! Explicit chunk size selection caveat: If chunk size is explicitly defined (either offline via the method described in this section, or at runtime using explicit scenario selection as described in section 3.6.3) *AND* chunk width/height does not divide evenly into IO widths/heights defined at run-time, then IO buffers must be allocated with the following properties:

1) IO buffer must be allocated with *at least* (chunk width – (IO width % chunk width)) elements of overflow space to the right of the core data region (for best performance, it is recommended that buffer span in bytes also be 16-byte divisible)
2) IO buffer must be allocated with *at least* (chunk height – (IO height % chunk height)) elements of overflow space below the core data region

For example, if an output chunk size is forced to be 8x8 and the associated output size is 333x222, the minimum required output buffer allocation would be:



If these requirements are not met, the data transfer to/from APEX memory may read/write beyond allocated buffer boundaries, resulting in undefined behavior.

Offline chunk size selection can be done using the following method:

```
SetInputChunkSize(<graph input port identifier>, <chunk width>, <chunk height>);
```

For example:

```
#include <ACF_Process_Desc_APU.hpp>
#include "myGraph_graph.hpp"

class myProcess_8x2_apu_process_desc : public ACF_Process_Desc_APU
{
public:

   void Create()
   {
      Initialize(mGraph, "myProcess_8x2");
      SetInputChunkSize("INPUT_0", 8, 2);
      SetInputChunkSize("INPUT_1", 8, 2);
   }

   myGraph_graph mGraph;
};
```

If this approach is taken, *ALL* non-fixed input chunk sizes must be specified. Furthermore, run-time I/O size flexibility will be reduced because only a single 'scenario' is analysed during the resolution phase. In the typical use case (i.e. when chunk size is NOT explicitly specified) the framework will analyse and store information pertaining to multiple scenarios (a 'scenario' being characterized by a base chunk size), allowing the ideal scenario to be selected at run-time based on process I/O sizes.

In the example above, the input chunk size above is chosen to be 8x2 for graph inputs 'INPUT_0' and 'INPUT_1' (8x2 is chosen arbitrarily for demonstration purposes). Typically for vector input types, **the input chunk width choice** should be made based on the following three factors:

1) The number of CUs (computational units) in the APU CU array (depends on APU configuration) and the maximum size of the input(s) that will be ultimately processed on the host-side
2) The worst case left and right spatial dependencies (if applicable). **Chunk width must be at least as wide as the worst case spatial dependency**. For example, if a graph contains a 3x3 kernel and a 9x9 kernel, chunk width must be at least 4 to satisfy the requirements of the 9x9 filter since it requires 4 samples beyond the left of the chunk and 4 samples beyond the right of the chunk.
3) To satisfy HW requirements, chunk width (*in bytes*) must be one of the following values: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 36, 40, 44, 48, 52, 56, 60, 64, 72, 80, 88, 96, 104, 112, 120, 128. If using an 'indirect' input the chunk width (*in bytes*) must be one of the following values: 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64.

ACF breaks input data down into tiles. Tiles are further divided into chunks. Each chunk maps to a CU in the APU. Chunk width must be selected such that the width of the input, when divided into chunks, fits into the CUs available in the APU (see section 3.7.1 for more information about chunks and tiling).

For example, if the input to the resolved process will ultimately be a 640x480 image and the APU consists of 32 CUs, chunk width must be set to at least 20 (640/32=20); anything smaller than 20 and the input tile will not 'fit' into the APU.

The **input chunk height choice** relates primarily to CMEM utilization. Larger chunk sizes mean more CMEM is consumed, especially as a graph gets complicated. As a general recommendation, keep the chunk height as small as possible during early development (it can be as low as 1).

3.5 ACF Offline Process Resolution (Step 4)

3.5.1 Overview

Offline resolution refers to the scripted process that takes the user-created inputs from steps 1, 2, and 3:

- kernel(s) + associated metadata (step 1)
- graph (step 2)
- process description (step 3)

and produces the following outputs:

- host-compatible 'handle' C++ class that encapsulates the resolved process and allows it to be instantiated, configured, and executed by a host-processor (see step 5)
- run-time binary that encapsulates the architecture specific machine code representing the ACF-generated processing pipeline. This binary is captured in a header file (*_APU_LOAD.h) that is included by the aforementioned handle class.

The following diagram depicts an offline resolution scenario assuming the identifier ***TEST***:

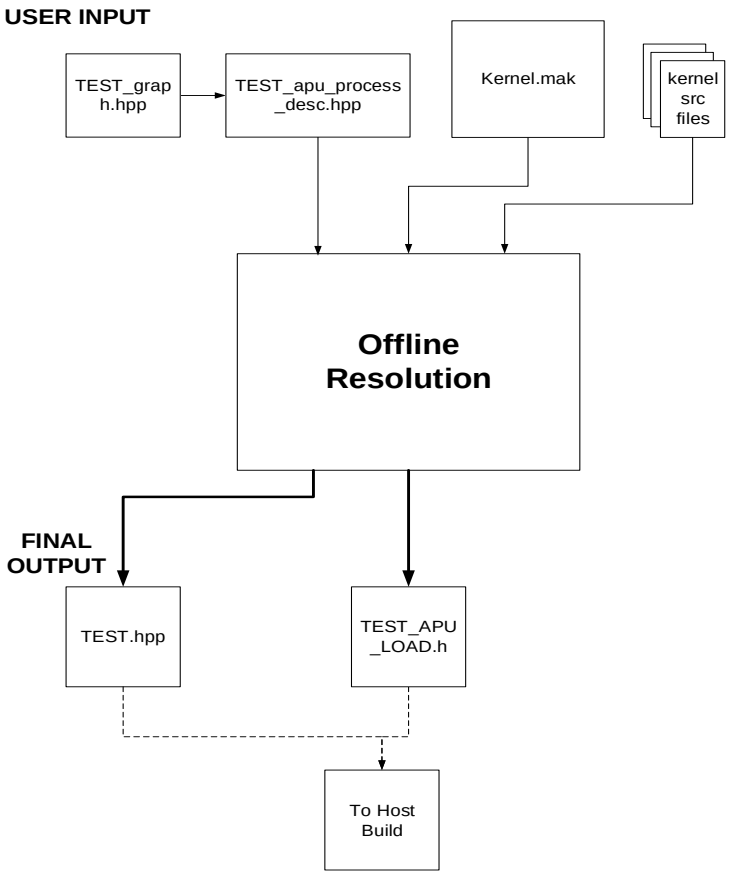


Figure 11 – Offline resolution process

ACF offline resolution is responsible for:

1. Kernel compilation and the generation of associated kernel libraries
2. Kernel metadata parsing and management
3. Graph resolution, and generation of the architecture-specific program encapsulating the processing pipeline
4. Generation of the aforementioned host 'handle' (TEST.hpp in above example)
5. Compilation of generated program (linked with required kernel libraries) and generation of final architecture specific binary load that encapsulates the processing pipeline (TEST_APU_LOAD.h in above example)

3.5.2 ACF Offline Error Messages

This section lists the possible ACF offline error messages that may be sent to standard output. Additional information is provided for those that are not deemed self-explanatory.

3.5.2.1 **ACF_Process_Desc_APU::WalkGraphForBaseED -> Base ED calculation for graph <graph identifier> is not stabilizing; aborting resolution**

This is likely caused by having kernel port sizes specified in such a way that it is impossible for the recursive base ED calculation to stabilize. E.g. Kernel A has two non-fixed outputs, one with ek(1,1) and one with ek(2,2), and these are connected to kernel B non-fixed inputs with ek(1,1) and ek(1,1) respectively.

3.5.2.2 **ACF_Process_Desc_APU::ConfigureLocalMemDesc(...) -> exceeded maximum number of memory descriptors (<ACF_PROCESS_MAX_NUM_LOCALMEMDESC>)**

At the time of writing ACF_PROCESS_MAX_NUM_LOCALMEMDESC is set to 500.

3.5.2.3 **ACF_Process_Desc_APU::AnalyzeScenarios() -> Graph <graph identifier> port <port identifier> is not fixed and the chunk size has not been set with SetInputChunkSize()**

This error will be output if the user chooses to call ACF_Process_Desc::SetInputChunkSize in their process descriptor for one non-fixed input port but neglects to call it for other non-fixed inputs. If any non-fixed inputs are being set with SetInputChunkSize, *ALL* non-fixed inputs must be set with SetInputChunkSize.

3.5.2.4 **ACF_Process_Desc_APU::AnalyzeScenarios() -> Issue encountered trying to calculate base eD for graph <graph identifier>; aborting resolution**

See 3.5.2.1.

3.5.2.5 **ACF_Process_Desc_APU::AnalyzeAndResolve() -> A viable scenario could not be found. Please see the analysis log file (/out/<process identifier>_analysis_log.txt).**

3.5.2.6 **ACF_Process_Desc::Initialize(...) -> failed**

3.5.2.7 **ACF_Process_Desc::Initialize(...) -> 'lProcessIdentifier' is either empty or does not have a meaningful value**

The select process identifier must be a string that is not an empty string or “”.

3.5.2.8 **ACF_Process_Desc::SetInputChunkSize(...) -> graph <graph identifier> input port <port identifier> has a fixed size and cannot be changed from (<port ek.x, ek.y>)**

3.5.2.9 **ACF_Process_Desc::SetInputChunkSize(...) -> graph <graph identifier> input port <port identifier> does not exist**

3.5.2.10 **ACF_Process_Desc::SetInputChunkSize(...) -> graph <graph identifier> input port <port identifier> ChunkSize is out of range; size values must be in the range [1:65535]**

3.5.2.11 **ACF_Process_Desc::SetInputChunkSize(<port identifier>, ...) -> process has not been successfully initialized**

3.5.2.12 **ACF_Process_Desc::FlagInputAsChunkBasedIndirect(...) -> graph <graph identifier> input port <port identifier> is not compatible with indirect mode; valid candidate must be a non-static vector port with no spatial dependencies**

3.5.2.13 **ACF_Process_Desc::FlagInputAsChunkBasedIndirect(...) -> graph <graph identifier> input port <port identifier> does not exist**

3.5.2.14 **ACF_Process_Desc::FlagInputAsChunkBasedIndirect(<port identifier>, ...) -> process has not been successfully initialized**

3.5.2.15 **ACF_Process_Desc::FlagInputAsVerticalSdOverlap(...) -> graph <graph identifier> input port <port identifier> does not exist**

3.5.2.16 **ACF_Process_Desc::FlagInputAsVerticalSdOverlap(<port identifier>, ...) -> process has not been successfully initialized**

3.5.2.17 **ACF_Process_Desc::CalcInputChunkSize(...) ChunkSize is out of range; size values must be in the range [1:65535]**

3.5.2.18 **ACF_Graph::AddInputPort(<port identifier>) -> error detected (see ACF_Node specific error above)**

The related ACF_Node specific error (typically output before this one) should provide more detailed information.

3.5.2.19 **ACF_Graph::AddOutputPort(<port identifier>) -> error detected (see ACF_Node specific error above)**

The related ACF_Node specific error (typically output before this one) should provide more detailed information.

3.5.2.20 **ACF_Graph::GraphPort(<port identifier>) -> <graph identifier> port <port identifier> cannot be found**

3.5.2.21 **ACF_Graph::KernelPort(<kernel identifier>, <port identifier>) -> port doesn't exist**

3.5.2.22 **ACF_Graph::KernelPort(<kernel identifier>, <port identifier>) -> kernel doesn't exist**

3.5.2.23 **ACF_Graph::AddKernel(<kernel identifier>, <kernel database identifier>) -> if a kernel has a non-fixed output (e.g. <non-fixed output port identifier>) it must have at least one non-fixed input**

3.5.2.24 **ACF_Graph::AddKernel(<kernel identifier>, <kernel database identifier>) -> kernel couldn't be found in the database**

3.5.2.25 **ACF_Graph::AddKernel(<kernel identifier>, <kernel database identifier>) -> unknown port type encountered**

3.5.2.26 **ACF_Graph::AddKernel(<kernel identifier>, <kernel database identifier>) -> unknown port data type encountered**

3.5.2.27 **ACF_Graph::AddKernel(<kernel identifier>, <kernel database identifier>) -> invalid e0/ek dimension(s) detected (must be non-zero)**

3.5.2.28 **ACF_Graph::AddKernel(<kernel identifier>, <kernel database identifier>) -> exceeded maximum allowable kernels per graph (<ACF_MAX_NUM_KERNELS_PER_GRAPH>)**

At the time of writing ACF_MAX_NUM_KERNELS_PER_GRAPH is set to 100.

3.5.2.29 **ACF_Graph::AddKernel(<kernel identifier>, <kernel database identifier>) -> memory allocation failure (mKernelArray)**

Internal memory allocation error.

3.5.2.30 **ACF_Graph::Connect(...) -> detected multiple connections to kernel(graph) <kernel/graph identifier> input(output) port <port identifier>; only a single connection is permitted**

3.5.2.31 **ACF_Graph::Connect(...) -> exceeded maximum number of connections per port**

At the time of writing ACF_MAX_NUM_FWD_DIR_CONNECTIONS_PER_PORT is set to 100.

3.5.2.32 **ACF_Graph::Connect(...) -> memory type (CMEM/DMEM) mismatch between <kernel/graph port identifier> port <src port identifier> and <kernel/graph port identifier> port <dst port identifier>**

3.5.2.33 **ACF_Graph::Connect(...) -> e0 mismatch between <kernel/graph port identifier> port <src port identifier> and <kernel/graph port identifier> port <dst port identifier>**

3.5.2.34 **ACF_Graph::Connect(...) -> invalid input/output connection between <kernel/graph port identifier> port <src port identifier> and <kernel/graph port identifier> port <dst port identifier>**

3.5.2.35 **ACF_Graph::Connect(...) -> 'lpSrcPort' and 'lpDstPort' are NULL**

3.5.2.36 **ACF_Graph::Connect(...) -> 'lpSrcPort' is NULL**

3.5.2.37 **ACF_Graph::Connect(...) -> 'lpDstPort' is NULL**

3.5.2.38 **ACF_Process_Desc_APU::Resolve() -> Graph has no input ports**

3.5.2.39 **ACF_Process_Desc_APU::Resolve() -> Graph has no output ports**

3.5.2.40 **ACF_Graph::VerifyPortConnections() -> graph input port ‘<port identifier>’ is not connected to anything**

3.5.2.41 **ACF_Graph::VerifyPortConnections() -> graph output port ‘<port identifier>’ is not connected to anything**

3.5.2.42 **ACF_Graph::VerifyPortConnections() -> kernel ‘<kernel identifier>' input port '<port identifier>' is not connected to anything**

3.5.2.43 **ACF_Graph::SetKernelPortOutputDelay(<kernel identifier>,<port identifier>) -> port doesn't exist**

3.5.2.44 **ACF_Graph::SetKernelPortOutputDelay(<kernel identifier>, <port identifier>) -> kernel doesn't exist**

3.5.2.45 **ACF_Node::AddInputPort(<port identifier>) -> the total number of ports (input + output) per node must not exceed (<ACF_MAX_NUM_PORTS_PER_NODE>)**

3.5.2.46 **ACF_Node::AddInputPort(<port identifier>) -> port identifier is not unique**

3.5.2.47 **ACF_Node::AddInputPort(<port identifier>) -> port identifier string is either empty or does not have a meaningful value**

3.5.2.48 **ACF_Node::AddInputPort(<port identifier>) -> memory allocation failure (mInputPortArray)**

3.5.2.49 **ACF_Node::AddOutputPort(<port identifier>) -> the total number of ports (input + output) per node must not exceed (<ACF_MAX_NUM_PORTS_PER_NODE>)**

At the time of writing ACF_MAX_NUM_PORTS_PER_NODE is set to 50.

3.5.2.50 **ACF_Node::AddOutputPort(<port identifier>) -> port identifier is not unique**

3.5.2.51 **ACF_Node::AddOutputPort(<port identifier>) -> port identifier string is either empty or does not have a meaningful value**

3.5.2.52 **ACF_Node::AddOutputPort(<port identifier>) -> memory allocation failure (mOutputPortArray)**

Internal memory allocation error.

3.5.2.53 **ACF_Node::SetIdentifier() -> identifier string is either empty or does not have a meaningful value**

3.5.2.54 **ACF_Node::InputPort(<port idx>) -> port doesn't exist.**

3.5.2.55 **ACF_Node::OutputPort(<port idx>) -> port doesn't exist.**

3.6 Configuring and Launching a Process on the Host (Step 5)

The final step occurs in the host environment and involves configuring and launching a resolved ACF process on APEX hardware.

3.6.1 ACF Host Interface and Example

As mentioned in section 3.5, the host-ready APU process is represented by a pair of generated header files:

<process_identifier>.hpp

<process_identifier>_APU_LOAD.h

An example of a host test stub that configures and invokes the resolved process (associated with the ongoing example 'myProcess') follows. The inputs to the function `myProcess_teststub` are of type `vsdk::UMat`.

```
#include <umat.hpp>
#include <myProcess.hpp>


int myProcess_teststub(vsdk::UMat lInput0,
                       vsdk::UMat lInput1,
                       vsdk::UMat lFilterCoef,
                       vsdk::UMat lOutput0)
{
   int lRetVal = 0;

   myProcess lProcess;

   lRetVal |= lProcess.Initialize();

   lRetVal |= lProcess.ConnectIO("INPUT_0", lInput0);
   lRetVal |= lProcess.ConnectIO("INPUT_1", lInput1);

   lRetVal |= lProcess.ConnectIO("INPUT_FLT_COEF_0", lFilterCoef);
   lRetVal |= lProcess.ConnectIO("INPUT_FLT_COEF_1", lFilterCoef);

   lRetVal |= lProcess.ConnectIO("OUTPUT_0", lOutput0);

   lRetVal |= lProcess.Start();
   lRetVal |= lProcess.Wait();

   return lRetVal;
}
```

A few notes about the above test code:

- The identifiers used with ConnectIO (e.g. `INPUT_0`, `INPUT_1`, `INPUT_FLT_COEF_0`, etc.) are the user-specified graph input/output identifiers selected during graph creation (see section 3.3).
- Always ensure that the Initialize() method Is called before a process is configured and launched for the first time. Initialize() does not need to be called again for subsequent launches of the same process as long as the process object has not been destroyed.

- Start() is a non-blocking call and you may perform other host-side processing in parallel with the APU process execution. Always make sure to (eventually) pair each call to Start() with a call to Wait().

Please see UG-10267-04 for a more detailed description of ACF_Process_APU.

3.6.2 Executing a Process with a Specific HW configuration (*ADVANCED*)

By default, a process will run on APU 0 with all available CUs, on APEX 0. The following method can be used to override the default configuration and specify the APEX HW a process will execute on.

```
SelectApuConfiguration(<apu configuration>, <apex id>);
```

Please see UG-10267-04 for a more detailed description of this ACF_Process_APU method and the configurations available.

3.6.3 Explicit Scenario Selection (*ADVANCED*)

If chunk size information is not specified offline in the process description (see section 3.4.2), the ACF offline 'resolution' phase will analyze and keep track of multiple scenarios (a scenario is uniquely identified by a base chunk size). By having multiple valid scenarios to choose from at run-time, ACF is able to choose the ideal scenario based on the actual I/O sizes and target APU configuration (information that is often only known at run-time).

In certain use cases, it may be desirable to select a specific scenario at run-time. This can be done with the following method. It forces the selection of the scenario whereby a specific port has a specified chunk size (an error will be returned if such a scenario does not exist). !!!**Note that when using SelectScenario(…), the target port identifier must refer to a port with *all* of the following properties: non-fixed & direct (i.e. not indirect) & non-static & vector**.

```
SelectScenario(<graph port identifier>, <chunk width>, <chunk height>);
```

Furthermore, once a scenario has been successfully selected, it is possible to query other graph ports to return the chunk size associated with each, if required.

```
QueryPortChunkSize(<graph port identifier>, <chunk width>, <chunk height>);
```

!!! Please review the 'Explicit chunk size selection caveat' in section 3.4.2 before using this functionality.

Please see UG-10267-04 for a more detailed description of these ACF_Process_APU methods.

3.7 Advanced ACF Functionality and Use Cases

This section discusses more advanced concepts and uses cases, with a focus on a high level understanding of how various port attributes map to actual data transfers between external memory and APEX local memory.

3.7.1 The Subdivision of Input Data: Vectorization vs. Tiling

It is important that the differences between **vectorization** and **tiling** be clearly understood within the context of ACF before proceeding with this section.

3.7.1.1 Vectorization

In the ACF context, **vectorization** refers to the subdivision of input data into smaller pieces (i.e. chunks) for the purpose of distribution across multiple processors to be processed in parallel (i.e. data level parallelism).

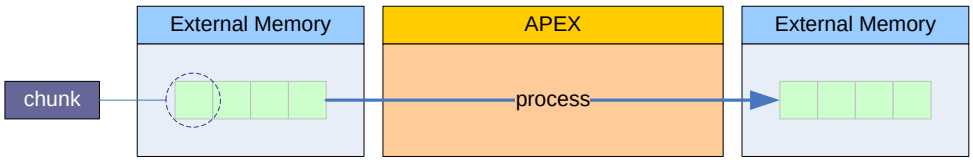


Figure 12 - Vectorization

3.7.1.2 Tiling

In the ACF context, **tiling** refers to the subdivision of input data into 'tiles' for sequential or *iterative* processing (a tile is a grouping of one or more chunks in a row).

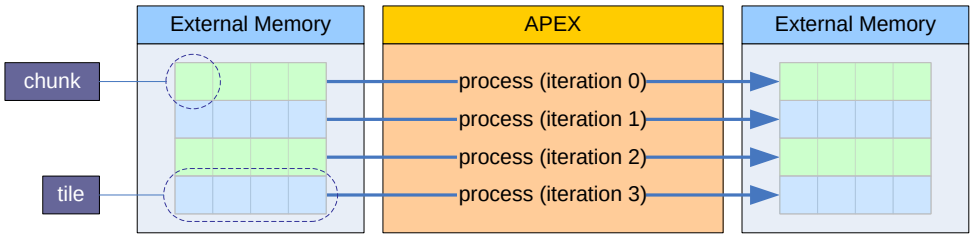


Figure 13 - Tiling

The need for tiling is in part a consequence of limited local APEX memory. For example, the APU has relatively small amounts of local memory. In typical use cases, input data sizes are much too large to fit entirely into CMEM (e.g. a megapixel image), so input data must be subdivided into tiles and moved into APU memory, processed, and moved out of APU memory in a producer/consumer fashion.

Tiling also improves parallelism and data locality. By breaking the processing into tiles and moving the input/output data to/from APU memory, ACF minimizes the costs associated with memory access latencies and data transfers by pipelining tile transfers with processing.

3.7.2 Attributes

3.7.2.1 **VEC**

By flagging an input port as a vector input, the framework is being told that the input data is a candidate for vectorization. This means that the framework is permitted to break associated input data into smaller pieces (chunks) and distribute the input data chunks across multiple processors for parallel processing.

In the APU case specifically, input data flagged as VEC is subdivided into chunks and distributed across the SIMD processing array.

3.7.2.2 **SCL**

By flagging an input port as a scalar input, the framework is being told that input data is ***not*** a candidate for vectorization (i.e. the data cannot be split into smaller pieces and distributed across multiple processors).

In the APU case specifically, input data flagged as SCL is written to APU DMEM. Note that scalar data may still be subject to tiling.

3.7.2.3 **(non-static)**

By flagging an input port as non-static, the framework is being told that input data is a candidate for tiling.

Input data transfers from external memory to local APEX memory occur tile by tile in an iterative fashion as determined by the total input size and the user-selected chunk size. Note that the number of iterations (i.e. the number of tiles) must be consistent across ALL non-static inputs.

Output data transfers from local APEX memory to external memory are handled in the same iterative fashion as input transfers.

3.7.2.4 **STATIC**

By flagging an input port as static, the framework is being told that input data should not be tiled and that a single local static APU buffer will be associated with this data (i.e. no circular buffering, dual or n-degree, will take place).

Static input data transfers from external memory to local APU memory ***occur only once prior to the commencement of any APU processing***. Such inputs are treated as monolithic data transfers. A kernel that has a static input can assume that the entirety of the static input data is available for reading at all times.

Static output data transfers from local APU memory to external memory ***occur only once following the completion of all APU processing***, and are treated as monolithic data transfers.

3.7.3 Understanding Attribute Combinations

It should now be clear what the VEC/SCL and (non-static)/STATIC attributes represent. This section will further clarify the different combinations and how they should be understood and used in a practical sense.

3.7.3.1 **(non-static) VEC**

The non-static vector attribute is used to indicate data that is both tileable and vectorizable. It should be used for 'large' inputs (e.g. image data) that can benefit from vectorization and parallel processing. It gives the framework maximum flexibility to take advantage of APEX processing resources.

Input data regions (and associated chunk sizes) can be 2D or 1D. In both cases the data will be subdivided into chunks and tiles in a 2D or 1D raster fashion (i.e. left to right, top to bottom).

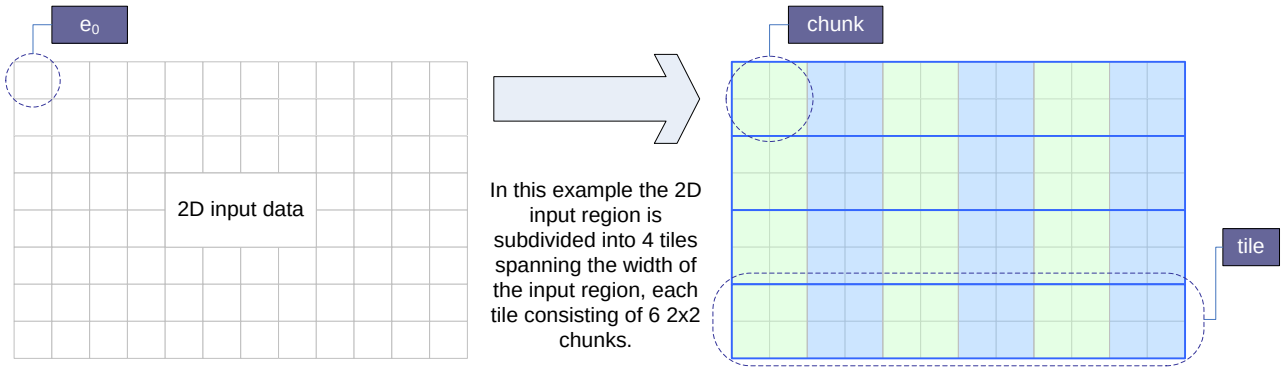


Figure 14 - Tiling of 2D data

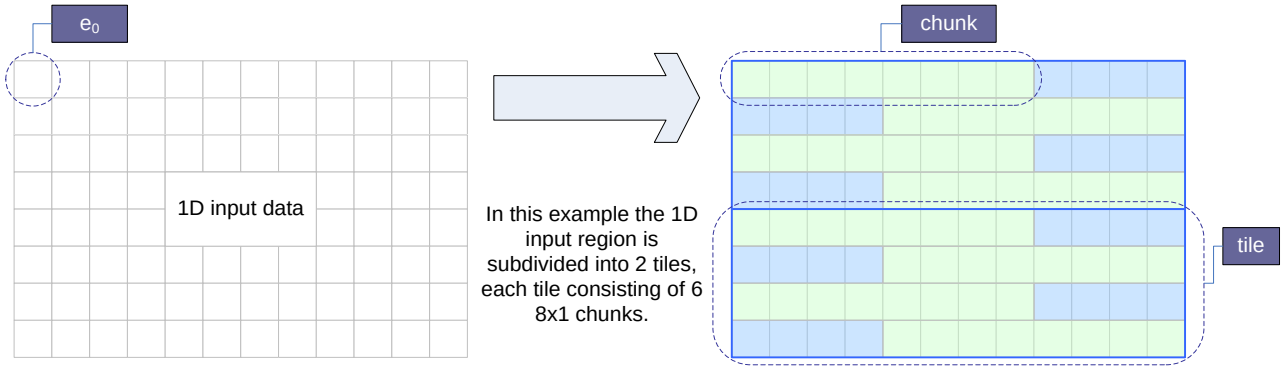


Figure 15 - Tiling of 1D data

3.7.3.2 **(non-static) SCL**

The non-static scalar attribute is tileable but not vectorizable and it can be used in the following situations:

- Bring in 'tiles' of data for scalar processing. Note that a tile will always consist of a single chunk in this case.

3.7.3.3 **STATIC SCL**

The static scalar attribute is used to indicate data that is neither tileable nor vectorizable. This type of port is useful when dealing with smaller amounts of input configuration/initialiation data (e.g. filter coefficients) or input/output ports that are associated with reduction operations. Please refer to section 3.7.4 for a more in-depth discussion of the reduction use case.

3.7.3.4 **STATIC VEC**

The static vector attribute is used to indicate data that is vectorizable but not tileable. This is a more advanced (and architecturally aware) use case, and it can be used in the following situations:

- Accumulate/preserve vector results between tiles as a means of partial reduction. A kernel is free to read from and write to a static vector buffer during each iteration.
- In a typical reduction use case a static vector output can be fed into a ‘reduction’ kernel for final reduction/processing.

3.7.4 Reduction Operations

This section will present a simple histogram use case in order to clearly demonstrate a vector reduction operation. Note that the histogram could be calculated using a scalar input if desired (with no reduction step necessary), but then it would not be taking advantage of APEX's parallel processing capabilities. This example focuses on an efficient non-static vector input + reduction scenario.

This section assumes the reader has a basic understanding of the histogram concept. For reference, an image histogram describes the tonal distribution of an image. For example, in an 8-bit greyscale image, each pixel can have a value that ranges from 0-255 (i.e. 256 possible values for each pixel). A 256-bit histogram for such an image would keep track of how many pixels in the image correspond to each value in the 0-255 range.

In this example the histogram kernel will be tabulating the frequency of 8-bit values ranging from 0 to 255 resulting in a final output list containing 256 32-bit values.

The graph representing the histogram scenario is as follows:

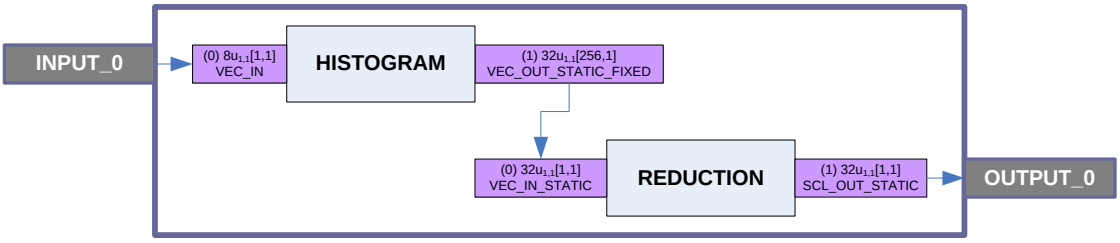


Figure 16 - Histogram graph

The histogram kernel input is a non-static vector, and the histogram kernel output is a static vector with a fixed size of 256x1. The output size is fixed because no matter what the input chunk size is, the output will always consist of 256 32-bit values.

Histogram output is a vector because each vector processor keeps track of its own 256-bin histogram result for the chunks of data that get assigned to it for processing (i.e. during each processing iteration, each vector processor will update its local cumulative histogram result based on the chunk of data it has been assigned).

Once processing has been fully completed (i.e. all tiles have been processed by the histogram kernel), *n* 256-bin results will exist across the vector processing array, where *n* is the tile width in chunks.

The final reduction step is required to reduce the *n* 256-bin results spread across the vector processing array into a single 256-bin scalar output. Notice that the reduction kernel has a static vector input and a static scalar output.

ACF provides a method *ACF_RET_VAR* to retrieve a variety of variables that are useful for cases like reduction (use of this method requires no additional #includes). See Table 3 below for a list of the variables that can be queried.

```
int16_t ACF_RET_VAR(ACF_VAR lVar);
```

ACF_VAR_FIRST_TILE_FLAG	Returns 1 if the chunk being processed belongs to the first tile, 0 otherwise.
ACF_VAR_LAST_TILE_FLAG	Returns 1 if the chunk being processed belongs to the last tile, 0 otherwise.
ACF_VAR_TILE_WIDTH_IN_CHUNKS	Returns the width of the current tile in chunks.
ACF_VAR_FIRST_CUID	Returns the ID of the CU containing the first chunk of a tile (a tile is mapped to an array of CUs with consecutive IDs). **Note that from the APU/kernel perspective, CU array indexing always starts at 0**; this remains true even if you select a different APU configuration as described in 3.6.2 (e.g. if you call SelectApuConfiguration(…) with ACF_APU_CFG__APU_1_CU_32_63_SMEM_2_3, CU array indexing from the APU's perspective will still start at 0, not 32). APU specific example: if **ACF_VAR_TILE_WIDTH_IN_CHUNKS** = 8 and **ACF_VAR_FIRST_CUID** = 1 then the tile is being processed by CUs 1 through 8 (inclusive) of the SIMD processing array.
ACF_VAR_NUM_INPUT_ITERATIONS	Returns the total number of input iterations (i.e. the number of input tiles that will be processed).
ACF_VAR_CU_ARRAY_WIDTH	Returns the width of the CU array. This may differ from 'tile width in chunks' because a tile may not span the entire CU array (e.g. CU array width = 64 and tile width in chunks = 60).

Table 3 - ACF variables (ACF_VAR)

The histogram kernel code can be found below. A few notes:

- The **ACF_VAR_FIRST_TILE_FLAG** variable is used to initialize the static vector output to zero *only once* (i.e. during the first iteration).
- Results will be accumulated on a per tile basis, generating a *running* result that is already a 'partial' reduction (i.e. it is a tile reduction). The reduction kernel will handle the vector reduction.

histogram.cpp

```
#ifdef ACF_KERNEL_METADATA

static KERNEL_INFO _kernel_info_histogram
(
   "HISTOGRAM",
   2,
   __port(__index(0),
          __identifier("INPUT_0"),
          __attributes(ACF_ATTR_VEC_IN),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d08u),
          __e0_size(1, 1),
          __ek_size(1, 1)),
   __port(__index(1),
          __identifier("OUTPUT_0"),
          __attributes(ACF_ATTR_VEC_OUT_STATIC_FIXED),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d32u),
          __e0_size(1, 1),
          __ek_size(256, 1))
);

#endif //#ifdef ACF_KERNEL_METADATA
```

```
#ifdef ACF_KERNEL_IMPLEMENTATION

void HISTOGRAM (kernel_io_desc lIn0, kernel_io_desc lOut0)
{
   vec08u* lpvIn0   = (vec08u*)lIn0.pMem;
   vec32u* lpvOut0  = (vec32u*)lOut0.pMem;
   int lStrideIn0   = lIn0.chunkSpan;
   int lStrideOut0  = lOut0.chunkSpan>>2;

   //initialize the static vector output buffer to zero
   if (ACF_RET_VAR(ACF_VAR_FIRST_TILE_FLAG))
   {
      for (int y=0; y<lOut0.chunkHeight; y++)
         for (int x=0; x<lOut0.chunkWidth; x++)
            lpvOut0[y*lStrideOut0+x] = 0;
   }

   for (int y=0; y<lIn0.chunkHeight; y++)
   {
      for (int x=0; x<lIn0.chunkWidth; x++)
      {
         vec08u lvBinIndex = lpvIn0[y*lStrideIn0+x];
         vec32u lvTmp = vload(lpvOut0, lvBinIndex);
         lvTmp += 1;
         vstore(lpvOut0, lvBinIndex, lvTmp);
      }
   }
}

#endif //#ifdef ACF_KERNEL_IMPLEMENTATION
```

Figure 17 - Histogram kernel

The reduction kernel implementation can be found below. A few notes:

- Since the primary graph input is tileable (i.e. it is non-static) the framework will invoke multiple processing iterations as required. Even though the reduction kernel will be invoked every iteration, the use of the **ACF_VAR_LAST_TILE_FLAG** variable ensures that the reduction kernel only performs the reduction operation on the final static histogram output vector once all other 'iterative' processing has completed.
- A for loop is used to iterative over the relevant processors to gather up the individual elements that comprise the vector result, making use of the **ACF_VAR_FIRST_CUID** and **ACF_VAR_TILE_WIDTH_IN_CHUNKS** variables. Within this loop, the vector result stored across the vector processing array (`lpvIn0`) is reduced to generate a final scalar result (`lpOut0`).

```
#ifdef ACF_KERNEL_METADATA

static  KERNEL_INFO _kernel_info_reduction
(
   "REDUCTION",
   2,
   __port(__index(0),
          __identifier("INPUT_0"),
          __attributes(ACF_ATTR_VEC_IN_STATIC),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d32u),
          __e0_size(1, 1),
          __ek_size(1, 1)),
   __port(__index(1),
          __identifier("OUTPUT_0"),
          __attributes(ACF_ATTR_SCL_OUT_STATIC),
          __spatial_dep(0,0,0,0),
          __e0_data_type(d32u),
          __e0_size(1, 1),
          __ek_size(1, 1))
);

#endif //#ifdef ACF_KERNEL_METADATA

#ifdef ACF_KERNEL_IMPLEMENTATION

void REDUCTION (kernel_io_desc lIn0, kernel_io_desc lOut0)
{
   if (ACF_RET_VAR(ACF_VAR_LAST_TILE_FLAG))
   {
      vec32u* lpvIn0 = (vec32u*)lIn0.pMem;
      int32_t* lpOut0 = (int32_t*)lOut0.pMem;
      int lChunkWidth = lIn0.chunkWidth;
      int lChunkHeight = lIn0.chunkHeight;
      int lChunkStrideIn0 = lIn0.chunkSpan>>2;
      int lChunkStrideOut0 = lOut0.chunkSpan>>2;

      //initialize the static scalar output buffer to zero
      for (int y=0; y<lChunkHeight; y++)
         for (int x=0; x<lChunkWidth; x++)
            lpOut0[y*lChunkStrideOut0+x] = 0;

      int16_t lFirstCuId         = ACF_RET_VAR(ACF_VAR_FIRST_CUID);
      int16_t lTileWidthInChunks = ACF_RET_VAR(ACF_VAR_TILE_WIDTH_IN_CHUNKS);

      for (int i=lFirstCuId;
               i<lFirstCuId+lTileWidthInChunks; i++)
      {
         for (int y=0; y<lChunkHeight; y++)
         {
            for (int x=0; x<lChunkWidth; x++)
            {
               lpOut0[y*lChunkStrideOut0+x] += vget(lpvIn0[y*lChunkStrideIn0+x], i);
            }
         }
      }
   }
}

#endif //#ifdef ACF_KERNEL_IMPLEMENTATION
```

Figure 18 - Reduction kernel

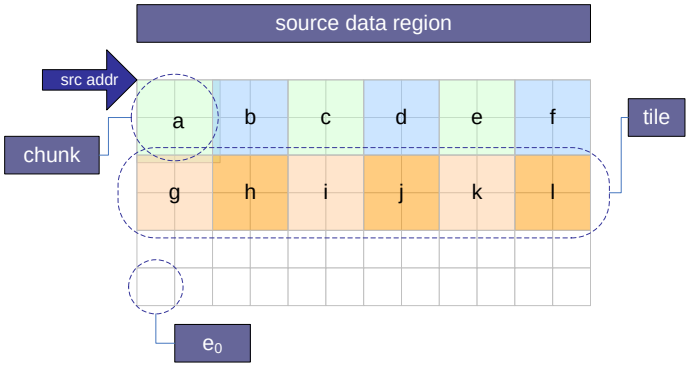
Additional Notes:

- It is advisable to keep references to ACF specific variables restricted to the ACF wrapper layer of your kernel implementations if possible (the above examples violate this recommendation for presentation purposes). From a development point of view, you will likely want to test and verify your kernel implementations (or at least as much of them as you can) alone in the APU simulator environment. Kernel files with references to ACF specific variables will not compile or function as expected since their meanings are tied to ACF.

3.7.5 Indirect Inputs

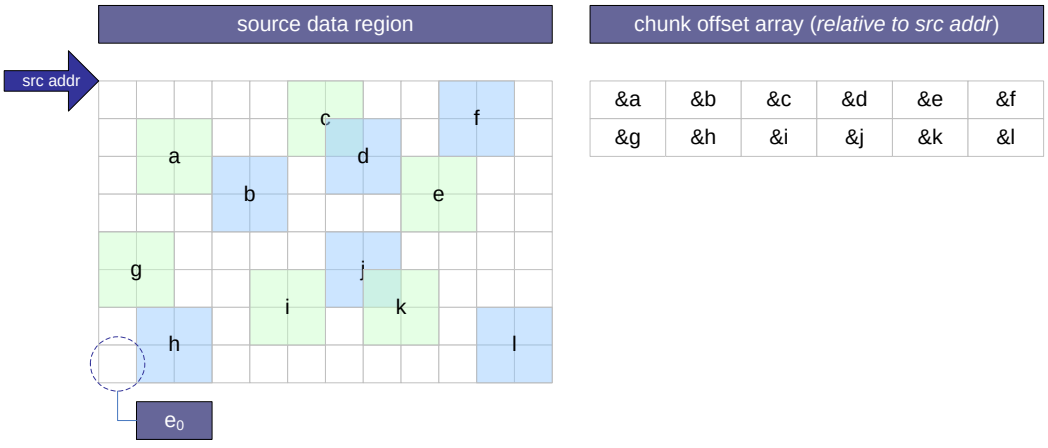
This section describes the indirect input functionality provided by ACF. Indirect input can be employed for those use cases where chunks of input of data residing in external memory do not adhere to a simple 1D or 2D raster pattern.

For reference, the following diagram illustrates a simple1D/2D raster data pattern where the chunks of data (a, b, c… j, k, l) are contiguous in memory.



In contrast to the raster pattern above, indirect input functionality allows the framework to construct tiles from chunks of data that are scattered throughout a source memory region. In addition to providing the source data, the user must also specify a chunk offset array. This 1D or 2D offset array contains a list of **byte offsets** (relative to the source data region starting point) that address the top left corners of the desired chunks.

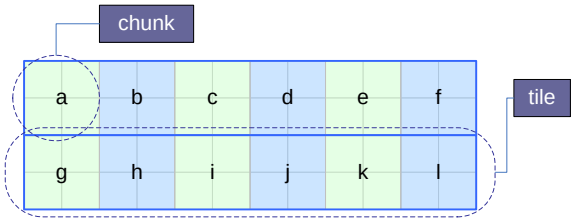
Consider the following example scenario where a user wishes to process 2 tiles, each consisting of 6 non-contiguous 2x2 chunks scattered throughout a source data region:



chunk offset array (*relative to src addr*)

&a	&b	&c	&d	&e	&f
&g	&h	&i	&j	&k	&l

Once the above information (i.e. the source data region and the chunk offset array) is provided to ACF, the 'effective' input from ACF's point of view would be as follows:



Note that indirect data transfer is only available for non-static vector inputs. Furthermore, the shape and size of the input (and any associated outputs) is determined by the shape and size of the chunk offset array and the associated chunk size. This can be seen in the example above - the chunk offset array is a 6x2 organization of chunk offsets, and the resulting 'effective' input is a 6x2 organization of 2x2 chunks.

In order to utilize this functionality, the following two steps are required:

1. During the process description step (see section 3.4), flag the desired input(s) as indirect using the following method:

```
int32_t FlagInputAsChunkBasedIndirect(std::string lInputPortIdentifier);
```

2. During the run-time IO configuration step (see section 3.6) use the *ConnectIndirectInput* method to provide ACF with both source data and the associated chunk offset array.

```
int32_t ConnectIndirectInput(std::string lPortIdentifier,
                             vsdk::UMat& lSrcData,
                             vsdk::UMat& lChunkOffsetArray);
```

Please refer to UG-10267-04 for more detailed descriptions of each of the aforementioned methods.

Known Limitations

1) If the chunk offset array is 2D the *width* of the offset array must be a multiple of 4 (i.e. the width must be a multiple of 4 offsets where each offset is associated with a chunk). If the chunk offset array is 1D, the *size* must be a multiple of 4. This limitation is related to the underlying HW.
2) To satisfy HW requirements, chunk width (*in bytes*) must be one of the following values: 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64

3.7.6 Region of Interest (ROI) Processing

3.7.6.1 Introduction

A region of interest (ROI) is defined as a 2D subset of data fully contained within a larger 2D 'parent' region. The ROI use-case is unique because it requires padding to come directly from the source data region for applicable edges.

Consider the following ROI:

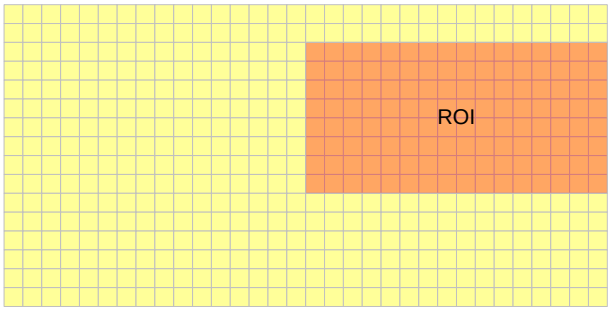


Figure 19 – A region of interest (ROI)

In the ROI depicted in Figure 19, there is source data available on the top, bottom, and left edges of the ROI. There is no source data available on the right edges of the ROI since it lines up with the edge of the parent data region.

Due to the availability of source data around the edges of the ROI, and the desire to take this source data into account during processing (especially important for ROI edges), special considerations need to be made when deciding how to manage ROI edge padding.

Assume that the ROI depicted in Figure 19 is fed into a single filter kernel with spatial dependencies defined as *sd(3,3,3,3)*. ROI edge padding will then be managed as depicted in Figure 20 below.

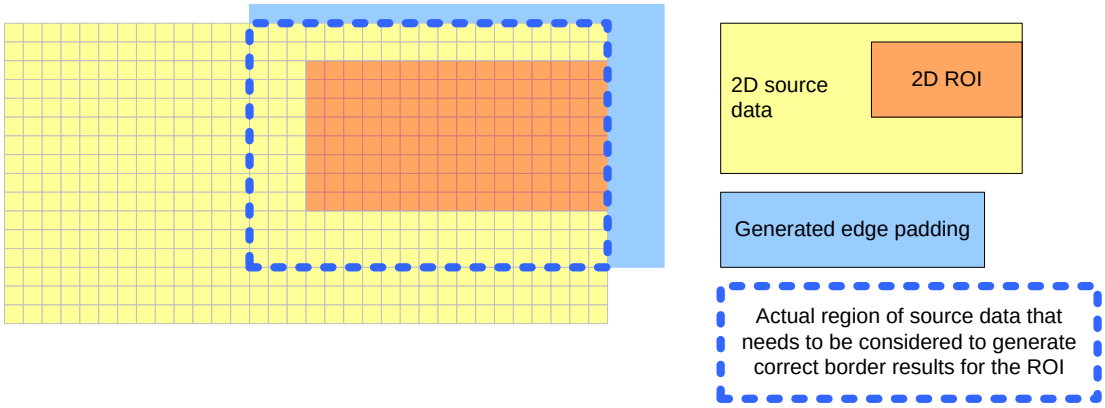


Figure 20 – ROI edge padding

As seen in Figure 20, up to three additional samples beyond each ROI edge must be considered. This is a direct result of the sd=(3,3,3,3) requirement. All 'generated' padding is handled by ACF (i.e. e0 replication) as described in section 3.2.2.3.

- Left and bottom edge padding comes entirely from the source data.
- Top edge padding consists of a mixture of source data and generated data (since 3 lines of padding are required but only two lines of source data are available).
- Right edge padding is entirely generated

3.7.6.2 Processing ROIs with ACF

From an interface point of view, configuring and connecting an ROI is managed at the host-level. You can specify and connect an ROI by using the the `ACF_Process::ConnectIO_ROI(…)` method (see UG-10267-04 for full interface details):

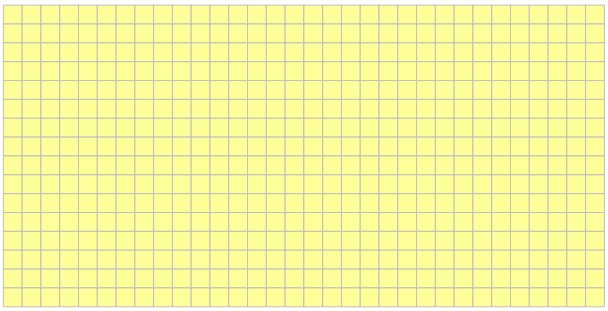
E.g. Assume we have a 640x480 input image, and that we want to process a 320x240 ROI that corresponds to the top left quadrant of the 640x480 source region.

```
//specify 640x480 input and output source regions
vsdk::UMat lInput = vsdk::UMat(480, 640, VSDK_CV_8U);
vsdk::UMat lOutput = vsdk::UMat(480, 640, VSDK_CV_8U);

MyProcess lProcess;
lProcess.Initialize();

lProcess.ConnectIO_ROI("INPUT", lInput, 0, 0, 320, 240);
lProcess.ConnectIO_ROI("OUTPUT", lOutput, 0, 0, 320, 240);
lProcess.Start();
lProcess.Wait();
```

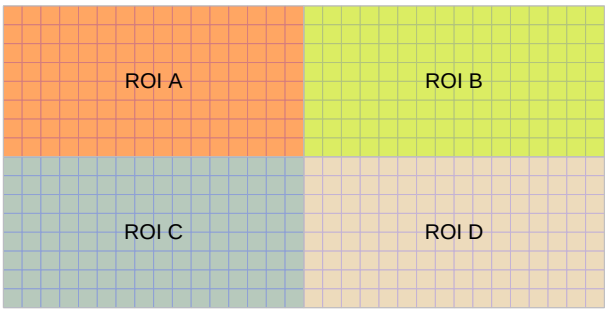
3.7.6.3 **Example ROI use Case**

One practical use case for ROI functionality is related to splitting a large input into multiple smaller inputs (i.e. multiple ROIs) and processing each ROI separately (while still producing the same result as if the original input had been processed as a whole). This sort of subdivision may be necessary if the combination of chunk size and input width results in an input tile that is too wide to fit into the available CU array.

Assume we want to process the following 32x16 input with *myFilterProcess*, which contains a single kernel with non-zero spatial dependencies.



In this example, assume the above input is subdivided into four 16x8 ROIs A, B, C, and D as illustrated below.



The host-code to process all four 16x8 ROIs is as follows:

```
vsdk::UMat lInput = vsdk::UMat(16, 32, VSDK_CV_8U);
vsdk::UMat lOutput = vsdk::UMat(16, 32, VSDK_CV_8U);

myFilterProcess lProcess;
lProcess.Initialize();
```

```
//top left 16x8 quadrant (i.e. ROI A)
lProcess.ConnectIO_ROI("INPUT", lInput, 0, 0, 16, 8);
lProcess.ConnectIO_ROI("OUTPUT ", lOutput, 0, 0, 16, 8);
lProcess.Start();
lProcess.Wait();

//top right 16x8 quadrant (i.e. ROI B)
lProcess.ConnectIO_ROI("INPUT", lInput, 16, 0, 16, 8);
lProcess.ConnectIO_ROI("OUTPUT ", lOutput, 16, 0, 16, 8);
lProcess.Start();
lProcess.Wait();

//bottom left 16x8 quadrant (i.e. ROI C)
lProcess.ConnectIO_ROI("INPUT", lInput, 0, 8, 16, 8);
lProcess.ConnectIO_ROI("OUTPUT ", lOutput, 0, 8, 16, 8);
lProcess.Start();
lProcess.Wait();

//bottom right 16x8 quadrant (i.e. ROI D)
lProcess.ConnectIO_ROI("INPUT", lInput, 16, 8, 16, 8);
lProcess.ConnectIO_ROI("OUTPUT ", lOutput, 16, 8, 16, 8);
lProcess.Start();
lProcess.Wait();
```

3.7.6.4 **Limitations**

- ROI functionality is limited to 2D non-static vector I/O port types.

- It is a requirement that both ROI coordinates and ROI size be divisible by the chunk size of the associated input/output port.

  For example, if the chunk size associated with the target input port is 8x4:

  Allowable ROI x coordinates include 0, 8, 16, 24, 32, etc.

  Allowable ROI y coordinates include 0, 4, 8, 12, 16, etc.

  Allowable ROI sizes include things like 320x240 and 240x100, but 300x200 would not be supported because 300 is not divisible by the chunk width of 8.

3.7.7 Interrupt Support

ACF internally makes use of APEX hardware interrupts to determine when process execution has completed. Assuming the target OS offers support for multi-threading (e.g. Linux), the default behaviour of `ACF_Process_APU::Wait()` is to perform a non-busy wait on an appropriate synchronization object (e.g. semaphore).

This default behaviour allows multiple ACF processes to be launched in multiple threads in parallel with ARM processing to make efficient use of both APEX and ARM resources.

In the bare-metal/no-OS use case, `ACF_Process_APU::Wait()` is still interrupt-driven, but it will ultimately be a polling wait.

Basic user-defined callback support is available via the overloaded `ACF_Process_APU::Start(...)` function (see document UG-10267-04-##-ACF_Reference_Guide.pdf for details):

```
int32_t Start(void (*lpCallback)(void* lpParam, int32_t* lpRetVal),
                    void* lpCallbackParam,
                    int32_t* lpCallbackRetVal);
```

As a general guideline, because `ACF_Process_APU::Wait()` already provides an abstracted interrupt-driven means of waiting for process completion, use of the user-defined callback should be reserved for more advanced/specific use cases. Even if a callback is specified, a call to `ACF_Process_APU::Start` must still be paired with a call to `ACF_Process_APU::Wait()`.

4 Appendix A (e_d)

4.1 Element<d>

Note: An ACF user doesn't need to configure Element<d>; this section is provided for the sake of a complete explanation.

The final element of this notation is an extension that allows the framework to express a 2D array of e_k's. Element<d> (or e_d) represents the fully described data chunk that is selected for kernel I/O. **e_d is chosen by the framework** using e_k as a guideline, to ultimately decide how input data will be broken up and fed into a kernel for processing. ***Note that a kernel developer should ensure e_k is as small as possible, because it gives the framework more flexibility when choosing e_d.***

Let e_d be written as:

$e_d = e_k$ {<num e_k in x dim>,<num e_k in y dim>}

Based on the decimate example above, the framework can choose an input $e_d = 8u_{1,1}$ [2,2]{4,4} (an 8x8 block of 8-bit data). As a consequence of this input selection, the decimated output $e_d = 8u_{1,1}$[1,1]{4,4} (a 4x4 block of 8-bit data).

Figure 21 - Decimate kernel (e_d)

4.2 Example with e_0, e_k, and e_d

Consider the following graph with all kernel I/Os expressed in e_k notation. Assume the user wants to process an 8x4 chunk of $8u_{4,1}$ data. Such an input to the 'YUV422 split' kernel would be expressed as $8u_{4,1}$ [1,1]{8,4}. The framework will set e_d dimensions to {8,4} for the 'YUV422 split' kernel to satisfy this input requirement. Based on this input requirement, how does the framework configure of e_d for the remaining two kernels?

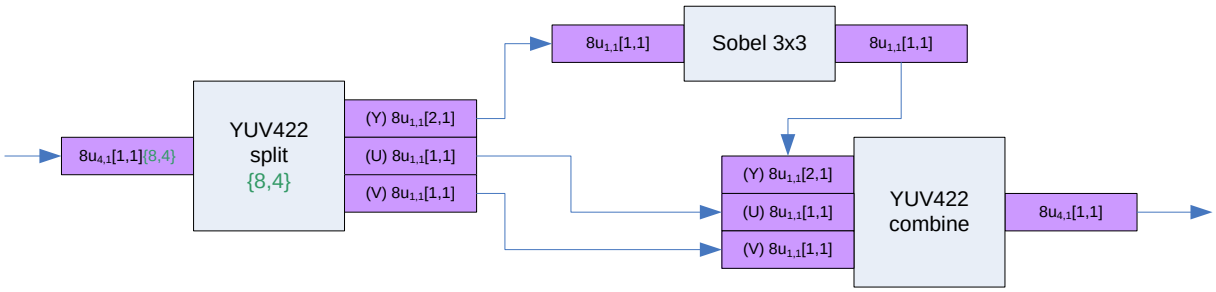


Figure 22 - YUV422 split/combine graph

The framework has already chosen e_d dimensions to be {8,4} for the 'YUV422 split' kernel to satisfy the input requirement. These same e_d dimensions are propagated to all 'YUV422 split' kernel ports, resulting in a luminance (Y) output $e_d = 8u_{1,1}$[2,1]{8,4} (for example).

The most noteworthy part of this graph is how the luminance (i.e. Y) output of the 'YUV422 split' kernel is being fed into the input of the 'Sobel 3x3' kernel. Note that even though the e_k of the luminance output ($8u_{1,1}[2,1]$) does not match the e_k of the Sobel input ($8u_{1,1}[1,1]$), the connection is allowed because $e_0=8u_{1,1}$ for both ports. While the difference in e_k dimensions does not preclude a connection, it does require that the e_d dimensions for the 'Sobel 3x3' kernel be configured differently than the 'YUV422 split' kernel.

The Sobel filter cannot directly accept an input with $e_d = 8u_{1,1}[2,1]\{8,4\}$ because the e_k dimensions do not match (it wants [1,1], not [2,1]). However, if the framework sets the e_d dimensions of the Sobel filter to be {16,4} everything matches up perfectly because $8u_{1,1}[2,1]\{8,4\}$ is equivalent to $8u_{1,1}[1,1]\{16,4\}$ from the e_0 point of view (i.e. both are 16x4 arrays of e_0).

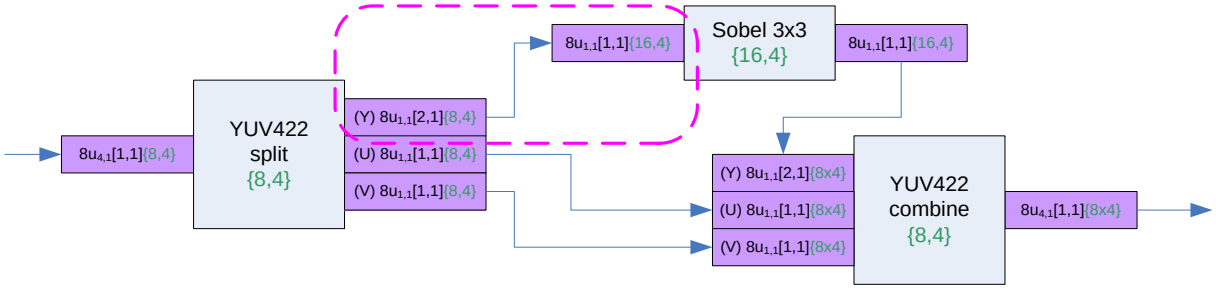


Figure 23 - YUV422 split/combine graph (e_d)