Vitis HLS Migration Guide

UG1391 (v2020.1) July 28, 2020

Revision History

Section	Revision Summary
07/28/2020 Version 2020.1	
Respin release.	N/A
06/03/2020 Version 2020.1	
Initial release.	N/A

Table of Contents

XILINX

Chapter 1

Migrating to Vitis HLS

When migrating a kernal module implemented in one version of Vivado HLS, it is essential to understand the difference between the Versions of HLS, and the impact that these differences have on the design.

Key Considerations:

- Behavioral Differences
- Unsupported Features
- Deprecated Commands

HLS Behavioral Differences

Vitis HLS brings some fundamental changes in the way HLS synthesizes the C code. These changes have implications on the application QoR. It is highly recommended to review the behavioral differences section before using the tool.

Default User Control Settings

The default global option configures the solution for either Vitis application acceleration development flow or Vivado IP development flow.

```
open_solution -flow_target <vitis | vivado>
```

This global option is replacing the old config option (`config_sdx`).

Vivado Flow:

Configures the solution to run in support of the Vivado IP generation flow, requiring strict use of pragmas and directives, and exporting the results as Vivado IP.

```
open_solution -flow_target vivado
```

Table 1: **Default Control Settings Table**

Default Control Settings	Vivado_hls	Vitis_hls
config_compile -pipeline_loops	0	64
config_export -vivado_optimization_level	2	0
set_clock_uncertainty	12.5	27%
config_export -vivado_optimization_level	20	255
config_interface -m_axi_alignment_byte_size	N/A	0
config_interface -m_axi_max_widen_bitwidth	N/A	0
config_export -vivado_phys_opt	place	none
config_interface -m_axi_addr64	false	true
config_schedule -enable_dsp_full_reg	false	true
config_rtl -module_auto_prefix	false	true
interface pragma defaults	*ip mode*	*ip mode*

Vitis Flow (Kernel Mode):

Configures the solution for use in the Vitis application acceleration development flow. This configures the Vitis HLS tool to properly infer interfaces for the function arguments without the need to specify the INTERFACE pragma or directive, and to output the synthesized RTL code as a Vitis kernel object file (.xo).

```
open_solution -flow_target vitis
```

Table 2: **Default Control Settings Table**

Default Control Settings	Vivado_hls	Vitis_hls
interface pragma defaults	*ip mode*	*kernel mode*(check default interfaces)
config_interface -m_axi_alignment_byte_size	N/A	64
config_interface -m_axi_max_widen_bitwidth	N/A	512
config_compile -name_max_length	256	255
config_compile -pipeline_loops	64	64
set_clock_uncertainty	27%	27%
config_rtl -register_reset_num	3	3
config_interface -m_axi_latency	0	64

Default Loop II Constraint Settings

In Vivado HLS, the default loop II constraint is set to 1, but in Vitis HLS it is set to auto. For example, if the tool could not achieve default II, it will try to achieve the best II possible.

Default Interfaces

The type of the interfaces that are created by interface synthesis depends on the data type of C argument, the default interface mode, and the interface directives. In Vitis HLS, the default interface changes depending on the data types used on the C arguments and configurations. The user selection of the `open_solution -flow_target <vitis | vivado>` will dictate the default interface settings.

The following figures shows the changes when the default interface protocol is enabled.

Argument type definitions (used in below tables):

- I: Input only (may only read from arg)
- O: Output only (may only write to arg)
- IO: Input & output (may read and write to arg)
- Return: Return data output
- Block: Block-level control
- D: Default mode for each typed.

***Note*:** If an illegal interface is specified, Vitis HLS issues a warning message and implements the default interface mode.

Vitis Flow (Kernel Mode)

If HLS is used in the Vitis flow, the tool will automatically set the following configurations.

```
open_solution -flow_target vitis
```

Table 3: **Argument Types**

Argument Type	Scalar		Pointer to an Array			Hls::stream
Interface Mode	**Input**	**Return**	**I**	**I/O**	**O**	**I and O**
ap_ctrl_none						
ap_ctrl_hs						
ap_ctrl_chain		D				
axis						D
m_axi			D	D	D	

The AXI4-Lite slave interface directive will change the behavior of the interface pragmas as shown below.

```
config_interface -default_slave_interface s_slave
```

Table 4: **Argument Types**

Argument Type	Scalar		Pointer to an Array			Hls::stream
Interface Mode	**Input**	**Return**	**I**	**I/O**	**O**	**I and O**
s_axi_lite	D	D	D	D	D	

***Note*:** These default interface pragma settings can be overridden by a user specified interface pragma.

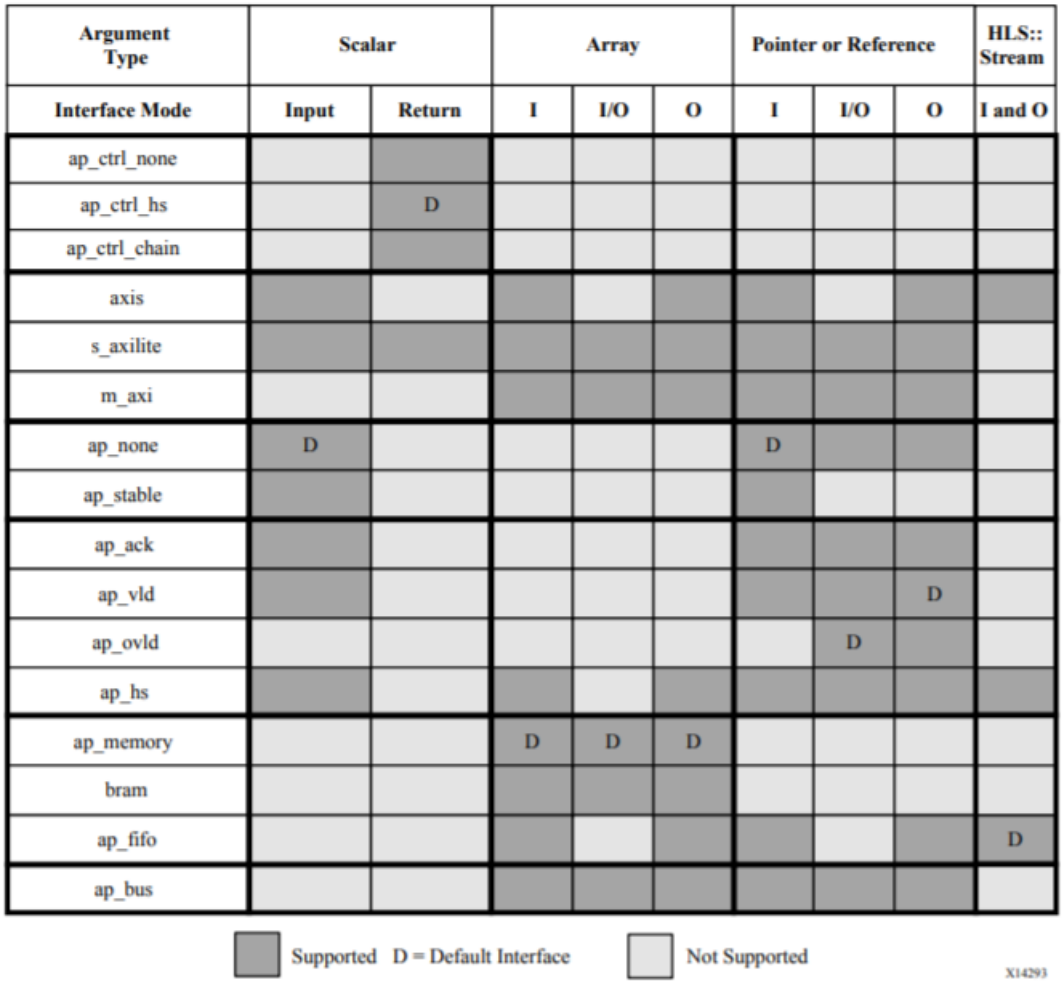
Vivado Design Flow

Vitis HLS can be used in standalone mode to create IP. The tool will run this flow by default for which it sets the following global options:

- `open_solution -flow_target vivado`
- `config_interface default_slave_interface off` (this means by default no AXI4-Lite interface pragma is automatically applied to the source code.)

In this case, the following default interfaces are applied.

Figure 1: **Argument Types**



Structs

Structs in the code, for instance internal and global variables, are disaggregated by default. They are decomposed into their member elements. The number and type of elements created are determined by the contents of the struct itself. Arrays of structs are implemented as multiple arrays, with a seperate array for each member of the struct. Structs in C/C++ are padded with extra bytes by the compiler for data alignment. In order to make kernel code in Vitis HLS compliant with gcc, structs in kernel code are padded with extra bytes. An example struct is shown in the Struct Padding and Alignment section:

Data Layout for Arbitrary Precision Types (ap_int library)

Data types in struct with custom data widths, such as `ap_int`, are allocated with sizes which are powers of 2. Vitis HLS adds padding bits for aligning the size of the data type to a power of 2.

In the following example, the size of `varA` in the struct will be padded to 8 bits instead of 5.

```
struct example  {
ap_int<5> varA;
unsigned short int varB;
unsigned short int varC;
int d;
};
```

TIP: *Vitis HLS will also pad the `bool` data type to align it to 8 bits.*

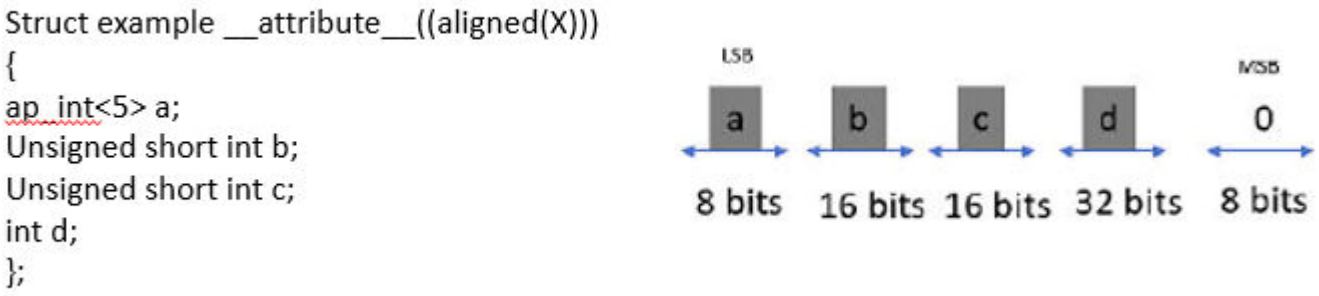
Struct Padding and Alignment

Structs in Vitis HLS can have different types of padding and alignment depending on the use of `__attributes__` or `#pragmas`. These features are described below.

- **Disaggregate:**

  By default, structs in the code as internal variables are disaggregated. Structs defined on the kernel interface are not. Disaggregated structs are broken up into individual elements as described in set_directive_disaggregate, or pragma HLS disaggregate. You do not need to apply the DISAGGREGATE pragma or directive, as this is the default behavior for the struct.

Figure 2: **Disaggregated Struct**



- **Aggregate:**

  This is the default behavior for structs on the interface, as discussed in Interface Synthesis and Structs. Vitis HLS joins the elements of the struct, aggregating the struct into a single data unit. This default is done in accordance with pragma HLS aggregate, although you do not need to specify the pragma as this is the default for structs on the interface. The aggregate process may also involve data padding for elements of the struct, to align the byte structures on a default 4-byte alignment.

  Note: The tool can issue a warning when bits are added to pad the struct, by specifying `-Wpadded` as a compiler flag.

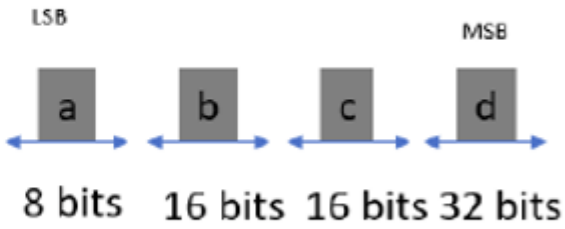
- **Aligned:**

  By default Vitis HLS will align struct on a 4-byte alignment, padding elements of the struct to align it to a 32-bit width. However, you can use the `__attribute__((aligned(X)))` to add padding to elements of the struct, to align it on "X" byte boundaries. In the figure below, the struct is aligned on a 2-byte boundary

IMPORTANT! *Note that "X" can only be defined as a power of 2.*

Figure 3: **Aligned Struct Implementation**

```
Struct example __attribute__((packed))
{
ap_int<5> a;
Unsigned short int b;
Unsigned short int c;
int d;
};
```



The padding used depends on the order and size of elements of your struct. In the following code example, the struct alignment is 4 bytes, and Vitis HLS will add 2 bytes of padding after the first element, `varA`, and another 2 bytes of padding after the third element, `varC`. The total size of the struct will be 96-bits.

```
struct data_t {
  short varA;
  int varB;
  short varC;
};
```

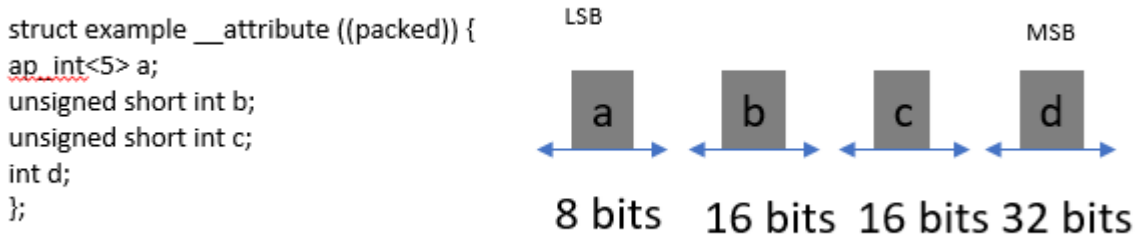
However, if you rewrite the struct as follows, there will be no need for padding, and the total size of the struct will be 64-bits.

```
struct data_t {
  short varA;
  short varC;
  int varB;
};
```

- **Packed:**

  Specified with `__attribute__(packed(X))`, Vitis HLS packs the elements of the struct so that the size of the struct is based on the actual size of each element of the struct. In the following example, this means the size of the struct is 72 bits:

Figure 4: **Packed Struct Implementation**




```
struct __attribute__((packed)) data_t {
   short varA;
   int varB;
   short varC;
};
```

Interface Bundle

The interface option contains a bundle option that groups function arguments into the AXI interface ports. The table below lists the behavioral changes if there is a mix of user defined/ none and default.

S_axilite Bundle

User-Defined and Default Bundle Names

- The tool will use the user-specified s_axi lite bundle name to create an axi lite port of the particular name.

```
void top(char *a, char *b, char *c, char *d) {
#pragma HLS INTERFACE s_axilite port=a bundle=terry
#pragma HLS INTERFACE s_axilite port=b bundle=terry
#pragma HLS INTERFACE s_axilite port=c bundle=stephen
#pragma HLS INTERFACE s_axilite port=d bundle=jim
}
```

- **Log File:** The synthesized axi lite port name can be viewed be viewed in the log file or RTL file.

```
INFO: [RTGEN 206-100] Bundling port 'return' to AXI-Lite port jim.
INFO: [RTGEN 206-100] Bundling port 'c' to AXI-Lite port stephen.
INFO: [RTGEN 206-100] Bundling port 'a' and 'b' to AXI-Lite port terry.
INFO: [RTGEN 206-100] Finished creating RTL model for 'example'
```

- When there is no bundle name specified the tool will use the default bundle name `-control` to create a axi lite port of that particular name.

```
void top(char *a, char *b, char *c, char *d) {
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c
#pragma HLS INTERFACE s_axilite port=d
}
```

- **Log File:** The synthesized axi lite port name can be viewed in the log file or RTL file.

```
Log fileINFO: [RTGEN 206-100] Bundling port 'a', 'b', 'c' to AXI-Lite
port control.
INFO: [RTGEN 206-100] Finished creating RTL model for 'example'.
```

- The tool will use a new default name `- control_r`. If the user mixes both the default bundle name `-control` and user-defined names to create an axi lite port.

```
void top(char *a, char *b, char *c, char *d) {
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c bundle=control
#pragma HLS INTERFACE s_axilite port=d bundle=control
}
```

- **Log File:** The synthesized axi lite port name can be viewed in the log file or RTL file.

```
INFO: [RTGEN 206-100] Bundling port 'c' and 'return' to AXI-Lite port
control.
INFO: [RTGEN 206-100] Bundling port 'a' and 'b' to AXI-Lite port
control_r.
INFO: [RTGEN 206-100] Finished creating RTL model for 'example'.
```

Maxi Bundle Options

- If the user specifies the config_interface - maxi_auto_max_ports = true, all the M-AXI interfaces that are not explicitly bundled will be mapped to individual axi mm port.

```
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
#pragma HLS INTERFACE m_axi port=a depth=50
#pragma HLS INTERFACE m_axi port=a depth=50
Log file
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem1' to
'm_axi'.
```

- If the user, sets the config_interface m_axi_auto_max_ports=false, then one of the following behavior will be true.
  - If the user maps all the MAXI interface pragmas to a bundle name - such as "terry". HLS generates a single MAXI port with the name - "terry".

```
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
Log file
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
```

  - If the user does not specify a bundle name on the interface pragma. HLS generates the default MAXI port name - gmem.

```
#pragma HLS INTERFACE m_axi port=a depth=50
#pragma HLS INTERFACE m_axi port=a depth=50
#pragma HLS INTERFACE m_axi port=a depth=50
Log file
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
```

  - If the user specifies the bundle name for some and ignores for the rest. HLS will bundle all the port with no bundle to the default port - gmem, and will use the use the user-specified bundle to generate the axi MM port.

```
#pragma HLS INTERFACE m_axi port=a depth=50 bundle=terry
#pragma HLS INTERFACE m_axi port=a depth=50
#pragma HLS INTERFACE m_axi port=a depth=50

Log file
INFO: [RTGEN 206-500] Setting interface mode on port 'example/terry' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem0' to
'm_axi'.
```

Interface Offset

The interface option contains an offset option that Controls the address offset in the AXI4-Lite (`s_axilite`) and AXI4 (`m_axi`) interfaces. The table below lists the behaviorial differences between the user specified/none and default.

Maxi Offset

Bundle Rules

The tool will use the user-specified offset if provided.

```
void top(char *a, char *b) {
#pragma HLS INTERFACE maxi port=a offset=direct
#pragma HLS INTERFACE maxi port=b offset=off
}
```

If the user does not specify the offset option.

- If default_slave_interface=off - offset will be direct.
- If default_slave_interface=s_axilite - offset will be slave.

```
void top(char *a, char *b) {
#pragma HLS INTERFACE maxi port=a bundle=my_maxi
#pragma HLS INTERFACE maxi port=b bundle=my_maxi
}
```

S_axilite Offset

S_axilite Scalar

In terms of bundling scalar ports/offsets to axilite adapter, there are the following three pragma scenarios:

Bundle Rules for Offsets

- **Fully Specified**: HLS will use the axilite adapter specified.
- **Not Specified**: By default for the Vitis flow, HLS will create an axilite adapter to transmit the scalar ports and offsets automatically.
- **Partially Specified**: Any unspecified offset or scalar will be bundled into an additional axilite adapter.

Memory Property on Interface

The storage_type option on the interface pragma or directive lets the user explicitly define which type of RAM is used, and which RAM ports are created (single-port or dual-port). If no storage_type is specified. Vitis HLS Uses:

- A single-port RAM by default.
- A dual-port RAM if it reduces the initiation interval or latency.

For the vivado flow, the user can specify a RAM storage type on the specified interface, replacing the old resource pragma with the storage_type .

```
#pragma HLS INTERFACE bram port = in1 storage_type=RAM_2P
#pragma HLS INTERFACE bram port = out storage_type=RAM_1P latency=3
```

AXI4-Stream Interfaces with Side-Channels

The AXI4-Stream with side-channel: Side-channels are optional signals which are part of the AXI4-Stream standard. The side-channel signals may be directly referenced and controlled in the C code using a struct, provided the member elements of the struct match the names of the AXI4-Stream side-channel signals. The AXI-Stream side-channel signals are considered data signals and are registered whenever `TDATA` is registered.

The user is not advised to use their own struct for AXIS side channels. It is recommended to use the class: `ap_axis_u.h` provided by the compiler.

```
#include "ap_axi_sdata.h"
#include "ap_int.h"
#include "hls_stream.h"

#define DWIDTH 32

typedef ap_axiu<DWIDTH, 0, 0, 0> trans_pkt;


extern "C" {
void krnl_stream_vmult(
                       hls::stream<trans_pkt> &b,
                       hls::stream<trans_pkt> &output) {
   #pragma HLS INTERFACE axis port=b
   #pragma HLS INTERFACE axis port=output
   #pragma HLS INTERFACE s_axilite port=return bundle=control

    bool eos = false;
vmult:
    do {
       #pragma HLS PIPELINE II=1
        trans_pkt t2 = b.read();

        // Packet for output
        trans_pkt t_out;

        // Reading data from input packet
        ap_uint<DWIDTH> in2 = t2.data;


        ap_uint<DWIDTH> tmpOut =  in2 *5;

        // Setting data and configuration to output packet
        t_out.data = tmpOut;
         t_out.last = t1.get_last();
         t_out.keep = -1; // Enabling all bytes

         // Writing packet to output stream
         output.write(t_out);
```

```
        if (t2.get_last()) {
            eos = true;
        }
    } while (eos == false);
}
}
```

Limitation

When using AXI4-Stream interfaces with side-channels and the function argument is a struct, Vitis HLS automatically packs all elements of the struct into a single wide-data vector. The interface is implemented as a single wide-data vector. This limitation in the tool does not let the user send a struct, such as RGBPixel across the axis interaface. The below workaround using the range operator should give the user the same behavior.

```
struct RGBPixel
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
    unsigned char a;

    RGBPixel(ap_int<32> d) : r(d.range(7,0)), g(d.range(15,8)),
b(d.range(23,16)), a(d.range(31,24)) {
#pragma HLS INLINE
    }

    operator ap_int<32>() {
#pragma HLS INLINE
        return (ap_int<8>(a), ap_int<8>(b), ap_int<8>(g), ap_int<8>(r));
    }

}__attribute__((aligned(4)));
```

config_rtl -module_auto_prefix

Behavior changes to *config_rtl -module_auto_prefix*

In Vivado HLS, when *config_rtl -module_auto_prefix* was enabled the top RTL module would have its name prefixed with its own name. In 2020.1 Vitis HLS this `auto` prefix will only be applied to sub-modules.

There is no change to the *-module_prefix* behavior, if this option is used the specified prefix value will be prepended to all modules including the top module. The *-module_prefix* option also still takes precedence over *-module_auto_prefix*.

```
# vivado HLS -2020.1 generated module names (top module is "top")
top_top.v
top_submodule1.v
top_submodule2.v

# Vitis HLS 2020.1 generated module names
top.v             <-- top module no longer has prefix
top_submodule1.v
top_submodule2.v
```

Dataflow

Support of std::complex:

In Vivado HLS, std::complex data type could not be used directly inside the DATAFLOW, because of multiple readers and writer issue. This multiple reader and writer issue is coming from the std class constructor being called to initialize the value. When this variable is also used inside the dataflow as a channel, it leads to the above issue. However, Vitis supports the use of `std::complex` with support of an attribute `no_ctor` as shown below.

```
// Nothing to do here.
void proc_1(std::complex<float> (&buffer)[50], const std::complex<float>
*in);
void proc_2(hls::stream<std::complex<float>> &fifo, const
std::complex<float> (&buffer)[50], std::complex<float> &acc);
void proc_3(std::complex<float> *out, hls::stream<std::complex<float>>
&fifo, const std::complex<float> acc);

void top(std::complex<float> *out, const std::complex<float> *in) {
#pragma HLS DATAFLOW
  std::complex<float> acc __attribute((no_ctor)); // here
  std::complex<float> buffer[50] __attribute__((no_ctor)); // here
  hls::stream<std::complex<float>, 5> fifo; // not here! (hls::stream has
it internally)

  proc_1(buffer, in);
  proc_2(fifo, buffer, acc);
  porc_3(out, fifo, acc);
}
```

Unsupported Features

The following features are not supported in this release.

IMPORTANT! *HLS will either issue a warning or error for all the unsupported features mentioned in this section.*

Top-level Function Arguments

Pragmas

- Pragma DEPENDENCE on an argument that also has an INTERFACE pragma with a m_axi bundle with 2+ ports is not supported.

```
void top(int *a, int *b) { // both a and b are bundled to m_axi port gmem
#prgama HLS interface m_axi port=a offset=slave bundle=gmem
#prgama HLS interface m_axi port=b offset=slave bundle=gmem
#pragma HLS dependence variable=a false
}
```

- Pragmas applied to top-level function arguments are not supported:
  - Array partition (Supports only dim =1, cyclic/block/complete)
  - Array reshape
  - Resource

Data Types

- The following data types on the top-level function arguments are not supported in this release.
  - enum or any use of enum (struct, array pointer of enum)
  - _Complex

- _Half, __fp16

Structs

- Pragmas such as dependence, resource, and interface applied to a particular struct field are not supported. The workaround is for the user to specify a disaggregate pragma and apply the dependence for a struct element.

```
Unsupported Form


struct ST {
int B[100];
int C;
};
struct ST aa;
#pragma HLS dependence variable=aa.B false
workaround
struct ST {
int B[100];
int C;
};
struct ST aa;#pragma HLS disaggregate variable=aa
#pragma HLS dependence variable=aa.B false
```

- Pragmas specified to a class/struct instance, which is the target to a method /variable inside them is unsupported. The workaround is to move this pragma inside the class/struct definition as shown in the following code. The limitation will be addressed in future releases.

```
class ST {

int B[100];

ST () {

#pragma HLS array_partition variable=B

#pragma HLS inline

}

};
```

HLS Video Library

- The `hls_video.h` for video utilities and functions has been deprecated and replaced by the Vitis vision library. See the Github Video Library Migration Guide for more details.

Arbitrary C Precision Types

Vitis HLS does not support Arbitrary precision C-types, user is advised to use C++ types with arbitrary precision.

Deprecated and Unsupported Tcl Command Options

Vitis HLS has deprecated and unsupported Tcl command Options.

Table 5: **Deprecated Tcl Commands Table**

Type	Command	Option	Vitis HLS 2020.1	Details
config	config_interface	-expose_global	Unsupported	
config	config_interface	-trim_dangling_port	Unsupported	
config	config_array_partition	-scalarize_all	Unsupported	
config	config_array_partition	-throughput_driven	Unsupported	
config	config_array_partition	-maximum_size	Unsupported	
config	config_array_partition	-include_extern_globals	Unsupported	
config	config_array_partition	-include_ports	Unsupported	
config	config_schedule	* (all options)	Deprecated	
config	config_bind	* (all options)	Deprecated	
config	config_rtl	-encoding	Deprecated	
config	config_sdx	-target	Deprecated	
config	config_flow	*	Deprecated	
config	config_sdx	-profile	Deprecated	Rename to `config_export-vivado_optimization_level`
config	config_sdx	-optimization_level	Deprecated	
directive	Clock	*	Unsupported	
directive	Data_pack	*	Unsupported	Use pack attribute.
directive	Function_instantiate	NA	Deprecated	
directive	Inline	-region	Deprecated	
directive	Array_map	*	Unsupported	
directive	Resource	*	Deprecated	Replaced by bind_op and bind_storage.
directive	Interface	*	Deprecated	
directive	Stream	-dim	Unsupported	
directive	Top	-name	Rename	

Table 5: **Deprecated Tcl Commands Table** *(cont'd)*

Type	Command	Option	Vitis HLS 2020.1	Details
config	config_dataflow	-disable_start_propagation	Deprecated	
config	config_rtl	-auto_prefix	Deprecated	Replaced by config_rtl -module_prefix.
config	config_rtl	-prefix	Deprecated	Replaced by config_rtl -module_prefix.
directive	Dependence	-class (array|pointer)	Deprecated	Workaround: Specify the variable name.
directive	Dependence	dependent (-true|false)	Deprecated	Workaround: Use "distance" value of >0 to replace "dependent=true".
directive	Dependence	-direction (RAW|WAR|WAW)	Deprecated	
directive	Dependence	-distance <integer>	Deprecated	
tcl	csim_design	-clang_sanitizer	Add/Rename	
tcl	export_design	-use_netlist	Deprecated	
tcl	export_design	-xo	Deprecated	
tcl	add_files	system-c	Unsupported	

1. **Deprecated:** A warning message for discontinuity of the pragma in the future release will be issued.
2. **Unsupported:** Vitis HLS errors out with a valid message.
3. ***:** All the options in the command .

XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *Vivado Design Suite User Guide: Designing with IP* (UG896)
2. *Vivado Design Suite: AXI Reference Guide* (UG1037)

Please Read: Important Legal Notices

Copyright